FORM

J.A.M.Vermaseren

August 8, 1989

Contents

Introduction

Nowadays the use of computerized symbolic manipulation methods is becoming more and more popular. The advantages offered by the automatic derivation of formulae are indeed great, not only for the very large formula crunching, but also for commonday 'small' problems. It can be expected that in the future tables of integrals may become superfluous in the same way that numerical programs and calculators made tables of logarithms superfluous. The field of formula manipulation is however split in two parts. On the one hand there is the class of small knowledge oriented problems for which the computer is very handy and time saving. On the other hand there are the very large problems that couldn't possibly be solved without lengthy computerized manipulations. This split is also reflected in the design of the symbolic manipulation programs. The 'well-known' popular programs are usually designed around a sophisticated set of instructions, some of which can only be executed properly if the entire formula exists inside the memory of the computer. This limits the size of those formulae. With such a program it is very easy to reduce the effective power of a computer to nearly zero when the formulae exceed a 'critical' size. Programs like Macsyma, Reduce, Maple and Mathematica fall in this class.

Programs that are designed for virtually infinite formulae use a completely different management of their memory. In those programs some sophisticated statements may be missing, although the tools may be present to execute the equivalent of such instructions on a step by step basis. The best known program of this type is Schoonschip which was designed to combine a very general basic instruction set and the power to deal with large for-

mulae. FORM belongs also to this class of programs. Its design allows it to work with formulae that contain millions of terms. On the other hand it has a reasonably extended instruction set so that it is also very convenient for many 'small' problems.

The basic philosophy and many of the instructions are derived from the model that Schoonschip provides. For years Schoonschip has been the only program that could deal with the demands posed by the large radiative correction computations in high energy physics. When it was programmed its author was envisaging formulae with about a hundred thousand terms. This is reflected in the fact that when the formulae become rather large Schoonschip will continue, but execution becomes much slower than necessary[1]. One of the new features of FORM is the management of really large formulae in a very efficient way. This is reflected in the impressive results of some speed tests.

FORM has been written in C to allow for easy installation on a large variety of systems. It will run both on personal workstations and large mainframe computers. This way the user may select the environment that suits his problem best. Normally the main limitation for very large programs will be the size of the available diskspace.

The syntax of FORM is a mixture of elements taken from Schoonschip, Fortran, C, Reduce and new inventions. The whole is designed to give an instruction set that is rather easy to remember. When the capacities of FORM are taken to their limits the user is advised to study the technical notes in the chapter on the setup file, as a knowledge of some of the internal workings allows the user to structure his programs such that they are performed in the fastest way. For casual use this will rarely be necessary.

The first version of FORM doesn't contain all planned instructions yet. It has been provided with the most needed instructions

[1] Recently this behaviour has been improved considerably.

and the mechanisms to make user libraries. Many big problems can be solved with it, even though the user may be required to think about the algorithms he should use. The examples section in this manual shows some algorithms that are particular to symbolic manipulations. This is anyway an important part of the manual. Symbolic programs like FORM are at their best when the intelligence of the user is combined with the power of the computer. In practice it is nearly impossible to provide algorithms that can always see the special cases that apply to a given problem. This holds especially for large research problems.

The manual is split in three parts. The first part is a tutorial in which the user is introduced in the language of FORM by means of a number of rather easy examples. The middle part contains the precise definitions of the language and the technical information that is needed to configure FORM to one whishes. The final part contains the more advanced examples that show the user a number of algorithms. This may enable him to work efficiently.

The author hopes to have contributed with FORM to the solution of many problems in the fields of mathematics, physics and engeneering.

Aknowledgements.
It has been a pleasure to enjoy the support of J.Smith, M.van der Horst and G.J.van Oldenborgh in various stages of the project.

Tutorial

As the best education is by means of examples we will start with a tutorial that contains a number of examples that will be simple at the beginning and rather involved as we use more features of FORM.

2.1 The first program

Tutorials of many languages start nowadays with a little program that prints the string 'Hello world'. The equivalent of this in formula manipulation is the evaluation of $(a+b)^2$. The FORM program for this is:

```
Symbols a,b;
Local f = (a+b)^2;
Print;
.end
```

Let us discuss the statements in order:

The first line of this little program is a declaration. It declares the objects a and b to be symbols. Symbols are the simplest objects in symbolic manipulation. They commute and they can have a power. In all respects they are scalar objects. Next we notice that the declaration was made with the word 'Symbols'. This doesn't mean that 'Symbols' is a protected word. FORM doesn't have any protected words. Each word that is legally formed can serve as a variable. The statements are formed by a keyword and then a follow up depending on the keyword. In this case the keyword 'Symbols' indicates that there will be a list of symbols.

The statement is concluded by a semicolon. The symbols can be separated by a comma or by blanks or by a combination of both. FORM considers blanks as relevant and replaces them by a comma, unless they are adjacent to a character that would indicate a separation already. Therefore the blank between 'Symbols' and 'a' might just as well have been a comma. On the other hand the blanks around the equals sign in the next line are irrelevant and serve for decoration purposes only. The last thing to be noticed in the first line is that 'Symbols' starts with a capital. This has been done to make the program look pretty. FORM is case insensitive with respect to its keywords and built in objects. It is on the other hand case sensitive with respect to the user defined quantities allowing the user full creativity when he chooses names. We will come back on this in the next section.

The second statement starts with the keyword 'Local' which indicates that f will be a 'local expression'. Expressions are the objects that are manipulated by FORM. They are individual formulae that contain terms. The observation that addition and subtraction are special commutative and associative operations allows us to define the concept of a 'term'. In the representation that it has at a given moment a term cannot be written as the sum or the difference of two or more constituents without splitting up the numerical coefficient of the term. Therefore $2*a$ can be a single term but $a+a$ is not. The sum of all terms make up an expression and expressions that can be manipulated are called active expressions. There can be many active expressions simultaneously as we will see in later sections. For the moment we have just one expression which is called 'f'. After the definition of an expression one has to tell FORM the starting contents of this expression, which is the formula we want to work out. One may note that a power is indicated by a carat: ^ as in many computer languages these days. The older Fortran notation of two stars ** is also accepted. When a formula is typed in one can use the rules

that are in use in all the major calculational languages, so there are no special rules with respect to the use of regular parentheses.

When the expression 'f' is being processed all parentheses will be evaluated. This means in our case that the square will be worked out. To make the result visible the user tells FORM that all expressions should be printed with the 'Print' command. If there are several expressions and the user wants to see only one of them it is possible to specify the expressions that should be printed after the keyword 'Print' (and a comma or a blank). In the current program that would be superfluous.

The final statement is not so much a statement but a directive. It tells FORM that there are no more statements and that execution can begin. There are several types of these directives that force execution of the statements that were given thus far. They all start with a period, followed by a keyword. Anything following the keyword is considered to be commentary, so a semicolon is not needed (it won't harm either). The allowable keywords are 'sort', 'store', 'global' and 'end'. Each of these statements that start with a keyword indicate the end of a 'module'. All statements in a module are translated first and then they are executed together. In our case there is only one module and since it is the last module its keyword is 'end'. When FORM encounters it it starts executing our little program.

The next complication is how to get FORM so far as to accept the above program. FORM is not interactive as many other symbolic systems. It is therefore necessary to prepare the program in a text editor. Suppose that we have done so and that the program has been put in a file example1. We can get FORM to execute this program by the simple command

```
form example1
```

This tells the operating system to start executing FORM and FORM will then recognize that its program should be found in

the file 'example1'. The result will be:

```
    Symbols a,b;
    Local f = (a+b)^2;
    Print;
    .end
```

```
Time =     0.06 sec     Generated terms =          3
              f         Terms left      =          3
                        Bytes used      =         54

   f =
      a^2 + 2*a*b + b^2;
```

The input is echoed which can be handy when the user wants to know what the program is doing (and for error messages). Then the run time statistics are printed. These include the execution time thus far (which depends of course on the computer), the number of terms that were generated, the number of terms in the output and the space these terms occupy together. Then the output is printed as we included a print statement. It is possible to tell FORM that it should not print the run time statistics but for the moment we will leave this option on. It allows us to see a little bit what is happening. When running large programs these statistics are quite vital because often it is possible to see from them that some mistake must have been made and the user can then kill the program, thereby saving much computer time.

It is now very simple to alter the little program somewhat to obtain the following:

```
Symbols a,b;
Local f = (a+b)^20;
Print;
.end
```

```
Time =     0.24 sec     Generated terms =         21
              f         Terms left      =         21
                        Bytes used      =        442

  f =
  a^20 + 20*a^19*b + 190*a^18*b^2 + 1140*a^17*b^3
  + 4845*a^16*b^4 + 15504*a^15*b^5 + 38760*a^14*b^6
  + 77520*a^13*b^7 + 125970*a^12*b^8 + 167960*a^11*b^9
  + 184756*a^10*b^10 + 167960*a^9*b^11
  + 125970*a^8*b^12 + 77520*a^7*b^13 + 38760*a^6*b^14
  + 15504*a^5*b^15 + 4845*a^4*b^16 + 1140*a^3*b^17
  + 190*a^2*b^18 + 20*a*b^19 + b^20;
```

This example shows one of the main applications of computer algebra: working out formulae that are beyond what is done easily by hand even though some people may write down the above result in less time than that it takes to type in the program that was used to generate this formula.

Let us study a few more variations of example 1:

```
Symbols a,b;
Local f = (a+b)*(a+b);
Print;
.end
```

```
Time =     0.07 sec     Generated terms =          4
              f         Terms left      =          3
                        Bytes used      =         54

   f =
      a^2 + 2*a*b + b^2;
```

In this case the answer is the same of course but because the input is given as the product of two subexpressions (formulae be-

tween parentheses), rather than a power of a single subexpression the number of generated terms is 4 rather than three. If we would do this with 20 brackets there would be `2^20` generated terms, even though the answer would have no more than 21 terms. One could argue that this is not efficient and indeed this is the case for this example. It comes in rather handy though when dealing with noncommuting objects. As a rule FORM never tries to interpret the right hand side of a statement until it needs it and even then it is inserted in its proper place before any interpretation takes place. At times this turns out to be enormously useful. If the user prefers the efficiency of the use of binomial coefficients he should use the notation with the power. Not using the power indicates that he doesn't want to use it and correspondingly FORM isn't going to use it anyway. This is one of the main properties of the language of FORM: "what you ask for is what you get".

The use of binomials in a power expansion is not an iron rule. The presence of noncommuting objects may prevent FORM from being too careless:

```
Functions a,b;
Local f = (a+b)^2;
Print;
.end

Time =     0.03 sec     Generated terms =          4
              f         Terms left      =          4
                        Bytes used      =         66

   f =
      a*a + a*b + b*a + b*b;
```

In this example a and b have been declared as functions. Normally functions will have arguments, but that is not necessary. A general property of regular functions is that they don't commute.

As no special precautions have been taken a and b should not commute and the binomial expansion cannot be used. The rule that is used by FORM is that the binomial expansion is used when there is at most one noncommuting object involved. This is demonstrated in the following example:

```
Symbols a,b,c;
Functions A,B,C;
Local f1 = (a+b+c)^5;
Local f2 = (a+b+C)^5;
Local f3 = (a+B+C)^5;
Local f4 = (a+(B+C))^5;
.end

Time =     0.13 sec     Generated terms =         21
             f1         Terms left      =         21
                        Bytes used      =        434

Time =     0.22 sec     Generated terms =         21
             f2         Terms left      =         21
                        Bytes used      =        510

Time =     1.04 sec     Generated terms =        243
             f3         Terms left      =         63
                        Bytes used      =       1786

Time =     1.28 sec     Generated terms =         63
             f4         Terms left      =         63
                        Bytes used      =       1786
```

The first and the second expressions can be evaluated with the binomial expansion. In the third expression this would cause errors as B and C should not commute. In the fourth expression we have helped FORM a little bit: by placing the parentheses the

object `(B+C)` becomes a single noncommuting object when the outer brackets are evaluated. these can then be evaluated with the binomial expansion. Then the powers of `(B+C)` are evaluated by brute force multiplication because this involves at least two noncommuting objects. It would of course be possible for FORM to place parentheses in the third expression, thereby economizing a great deal, but that would violate the principle of not touching the input. If the user likes to economize he can help FORM by placing the parentheses himself.

2.2 More types of variables

In the previous section we encountered symbols and functions. One may wonder what other types of variables there exist in FORM. First of all there is a class of functions that commute with everything. These are the 'commuting functions', indicated either by 'commuting' or by 'cfunctions'. A typical example of such a function is a factorial, or more general any function that results in a numerical value. When such a function has no arguments it behaves very much like a symbol. There is one exception: when functions are printed they have no power so the commuting function 'f' to the power 4 is printed as `f*f*f*f`. Functions can have any number of arguments. Arguments are separated by comma's:

```
CFunctions f,g,h;
Symbols x,y;
Local F =
+ g(x)*f(y)
- f((x+y)^2,x,y)
+ h(x,,y);
Print;
.end
```

```
Time =      0.10 sec     Generated terms =          6
               F         Terms left      =          3
                         Bytes used      =        222

   F =
   - f(x^2 + 2*x*y + y^2,x,y) + f(y)*g(x) + h(x,0,y);
```

We may notice here several things: First FORM has changed the order in which we specified the terms. This is a necessary property: in order to make sure that no term occurs twice the

output is brought into a unique form. It may happen that this form is not what the user expected. We will see later how we can exercise a little control over the exact form of the output.

The square inside the first argument of f has been worked out. This is why there are 6 generated terms in the statistics. Terms that are generated inside function arguments are also counted for generation purposes. They are not counted in the 'terms left' category.

Also the order of f and g has been changed to obtain the second term in the output. This can be done because the functions have been declared as commuting functions.

The empty argument in h has been replaced by a zero. In FORM empty arguments and arguments that are zero are identical. This may not be very elegant from the mathematical viewpoint but it should not cause any serious problems.

The two occurrences of f have a different number of arguments. This is a general feature: there is no check on the type and the number of the arguments of a function until something has to be done with the function. When such is the case only those occurrences that have the right number and the right types of arguments will be processed. The rest will be left untouched.

Many computations in science involve messy vector and tensor manipulations. To speed such computations up FORM is equipped with the variable types 'Index' and 'Vector'. Indices are quantities that can either be a formal integer number like '1' or '2', or a symbolic index like 'mu'. Vectors can occur as an object with an index like in 'p(mu)', a function argument or in so called dotproducts. Let us look at some examples:

```
Indices mu,nu;
Index alpha;
Vectors p,q;
CFunction f;
```

```
L   F1 = (p(mu)+q(mu))*(p(nu)+q(nu));
L   F2 = (p(mu)+q(mu))*(p(mu)+q(mu));
L   F3 = f(alpha,q+p)*p(alpha);
print;
.end

Time =      0.11 sec     Generated terms =          4
              F1         Terms left      =          4
                         Bytes used      =         82

Time =      0.13 sec     Generated terms =          4
              F2         Terms left      =          3
                         Bytes used      =         56

Time =      0.15 sec     Generated terms =          3
              F3         Terms left      =          1
                         Bytes used      =         64

   F1 =
    p(mu)*p(nu) + p(mu)*q(nu)
    + p(nu)*q(mu) + q(mu)*q(nu);

   F2 =
    p.p + 2*p.q + q.q;

   F3 =
    f(p,p + q);
```

In the first expressions we see that the output contains vectors of mu and nu. In the second expression FORM made an assumption which is called the 'Einstein summation convention': indices that occur twice inside the same term are considered to be summed over. In the case of `p(mu)*p(mu)` that means that

we obtain the dotproduct `p.p`. The order of the vectors in a dotproduct is considered to be unimportant. FORM assumes that vectors are commuting. Therefore `p.q` and `q.p` have been added. When the user doesn't like this he can use functions that don't commute. The penalty for the use of functions is a considerable slowdown of the execution.

In the third expression we see that the second argument of f has been rearranged again. We see also that alpha has disappeared from the expression altogether. When an index is summed over and in one of its occurrences it is the argument of a vector we may as well put the vector at the place of the other occurrence. This is called 'Schoonschip notation'. We may loose some information this way because we cannot tell from the output whether f is linear in its first argument because there used to be a contracted index. On the other hand: this is usually clear from its use.

There is something more to be noticed in this program: The declaration of the indices was done once with the keyword 'Indices' and once with the keyword 'Index'. This is a property of all declarations: both the plural and the singular is accepted. There is actually an even greater freedom in the use of keywords: most keywords are already recognized if the first one or two characters are given. This means that FORM would have recognized 'I' as 'Indices', 'S' as 'Symbols' and 'L' as 'Local'. For the clarity of the examples we will not use this abbreviated form of the commands in the tutorial part of the manual. In the later parts we will use it regularly. The user may notice that he himself will start doing this too by the time he gets some experience.

```
Index mu;
CFunction f;
Vector p;
Local F = p(mu)*p(1)
+f(p,mu,1);
```

```
print;
.end

Time =     0.07 sec     Generated terms =          2
              F         Terms left      =          2
                        Bytes used      =         82
     F =
     p(1)*p(mu) + f(p,mu,1);
```

In this example we see the use of the 'fixed index' 1. A fixed index is an index that will not be summed over. It has a numerical representation that must be integer, nonnegative (zero is allowed) and less than a given maximum (usually 128). This maximum can be altered if the need arises (see the chapter on the setup file). There is one problem with respect to fixed indices: It is impossible to see whether the third argument of f is a fixed index or a regular number. Usually this causes no problems as long as the user doesn't define f in such a way that both numbers and indices can occur at the same position in the argument field of a function.

One may have noticed also that we haven't specified whether indices are upper or lower indices. This has a profound reason: As long as there are no external indices left after a computation all indices have been contracted. In that case one index must have been an upper index and the other a lower index and no harm is done. Things become more dangerous when the user defines quantities as `(p(mu))^2` which would be replaced by `p.p`. In this case the user played a dirty trick with the notations if indeed he meant two upper or two lower indices. Until we start using conjugations there should be no problems with the choice of the metric as long as the user gives a proper input. It should then also be clear what the type of the index in the output is. In the case that things become too complicated there are always

(less efficient) solutions: using `f(mu,up)` and `f(mu,down)` with up and down two symbols and using upper and lower case characters for the vectors, depending on whether their indices are upper or lower indices gives all the notational freedom that is needed. From now on we will ignore the problem of the metric as for nearly all computations the FORM conventions are metric independent.

```
Symbols x,n;
Index mu=0,nu,la=3,ka=n;
Vector p;
L F = p(mu)*p(mu)
+ d_(mu,mu)
+ d_(nu,nu)*x
+ d_(la,la)*x^2
+ d_(ka,ka)*x^3;
print;
.end

Time =     0.07 sec     Generated terms =          5
              F         Terms left      =          5
                        Bytes used      =         90

   F =
      x^3*n + 3*x^2 + 4*x + p(mu)*p(mu) + d_(mu,mu);
```

In the above example we see how we can manipulate the dimension over which an index is summed. The dimension can be specified by the '=dimension' in the declaration after the name of the index. Any nonnegative integer less than the maximum number (usually 10000) is allowed. In addition the name of a symbol is allowed. When the dimension is given as zero the index will not be summed over. To demonstrate this we have introduced the built in object d_ which is the Kronecker delta. Its properties

are known to FORM, so when an index is summable and occurs more than once of which at least once in a Kronecker delta there will be some action. When both occurrences are inside the same delta the delta will be replaced by the dimension of the object. This is 3 for 'la', n for 'ka' and 4 for 'nu', because 4 is the default dimension. No replacement is made for `d_(mu,mu)` because mu isn't summable. It is assumed that all vectors have the default dimension. This default dimension can be changed:

```
Dimension 3;
Indices mu,nu=4,ka;
Symbol x;
L F =
  d_(mu,mu)
+ d_(nu,nu)*x
+ d_(mu,nu)*d_(mu,nu)*x^2
+ d_(nu,ka)*d_(nu,ka)*x^3;
print;
.end

Time =     0.07 sec     Generated terms =          4
              F         Terms left      =          4
                        Bytes used      =         58

   F =
      3 + 3*x^3 + 4*x^2 + 4*x;
```

The dimension statement allows the user to set the default dimension to any value that is allowed as a dimension (so also to a symbol). We see this clearly in the summation over mu and nu. The last two terms show a problem with respect to working with mixed dimensions. There is no check on whether the indices in a Kronecker delta have the same dimension. The final value of the `x^2` and `x^3` depends on the order in which the indices

were declared! One could argue that FORM should give an error message here, but there are cases in which –with the right amount of care– it can be very handy to allow such ambiguous notation.

The last type of variables is the 'set'. We will discuss it when we need it, because for a profitable use of it we need some more capabilities of FORM.

We will finish this section with defining what is a proper name. A name can have any number of characters and should be composed of alphanumeric characters of which the first must be an alphabetic character. Underscores, dollar signs etc are not allowed in user defined names. The built in names all end in an underscore, which means that there cannot be a conflict in the choice of names. In addition to this regular type of names there is a second type of names which was suggested by E.Remiddi: A name can consist of a matching pair of straight braces `[ ]` with any characters in between as long as these characters don't upset the fact that the outer braces match. This means that `[$^*?=+]` is a legal name. Also `[x+a]` is a legal name for FORM. To us this name may have some extra meaning but for FORM it is just a name as would be the name `xplusa`. This name convention has the great benefit of allowing FORM to work with single objects while it is clear for the user that this object stands in reality for something much more complex.

There is one exception to the rule that names can have an arbitrary number of characters: names of expressions are limited to 16 characters at most.

```
Symbol [p.p+m^2],m;
Vector p;
Local F1 = m/[p.p+m^2];
Local F2 = m/(p.p+m^2);
print;
.end
```

```
Time =     0.04 sec     Generated terms =          1
             F1         Terms left      =          1
                        Bytes used      =         22

Time =     0.06 sec     Generated terms =          3
             F2         Terms left      =          1
                        Bytes used      =         60

   F1 =
      [p.p+m^2]^-1*m;

   F2 =
      1/(m^2 + p.p)*m;
```

If we are not planning on doing much with the denominator in the above example it is much easier to define it as a single symbol. The statistics show that FORM has to generate more terms internally to work with F2 in which the contents of the denominator are not kept hidden from FORM. In general it is much faster to work with denominators that are single symbols and specify the relevant operations when they are needed than to let FORM have a go at them with general algorithms. Actually version 1.0 doesn't contain any algorithms at all to work with composite denominators. It just tolerates their presence.

2.3 The first substitutions

In the previous examples we saw the definition of expressions, but nothing was done with them. Symbolic manipulation would not be of much use if we would not have some way of altering formulae by means of operations and substitutions. Let us look at some of those:

In the first example we have a quadratic polynomial and demand that this polynomial and its derivative have common solutions. The condition that should be fulfilled such that two polynomials have common solutions has been written down by Sylvester in the form of a single determinant of size (m+n) in which m and n are the degrees of the two polynomials:

This means that we have to compute a determinant (3 by 3 in this example)

```
*
*   Condition for the simultaneous solution of
*   a quadratic equation and its derivative.
*   The equation is a2*x^2+a1*x1+a0 = 0
*
*   We need:
*           |  a0   a1   a2  |
*           |  a1 2*a2   0   |
*           |  0    a1 2*a2  |   =  0
*
Symbols x,a0,a1,a2;
CFunction M;
Indices m1,m2,m3;
Local F = e_(1,2,3)*e_(m1,m2,m3)*
          M(1,m1)*M(2,m2)*M(3,m3);
```

```
    contract;
    print;
    .sort

Time =     0.16 sec     Generated terms =          6
              F         Terms left      =          6
                        Bytes used      =        662

   F =
   M(1,1)*M(2,2)*M(3,3) - M(1,1)*M(2,3)*M(3,2) - M(1
   ,2)*M(2,1)*M(3,3) + M(1,2)*M(2,3)*M(3,1) + M(1,3)
   *M(2,1)*M(3,2) - M(1,3)*M(2,2)*M(3,1);

    id  M(1,1) = a0;
    id  M(1,2) = a1;
    id  M(1,3) = a2;
    id  M(2,1) = a1;
    id  M(2,2) = 2*a2;
    id  M(2,3) = 0;
    id  M(3,1) = 0;
    id  M(3,2) = a1;
    id  M(3,3) = 2*a2;
    print;
    .end

Time =     0.40 sec     Generated terms =          3
              F         Terms left      =          2
                        Bytes used      =         42

   F =
      4*a0*a2^2 - a1^2*a2;
```

The determinant is computed by noticing that determinants

are basically obtained via contractions of Levi-Civita tensors. A Levi-Civita tensor is a totally anti-symmetric tensor with values 1,0,-1. It is represented in FORM by the commuting function e_. The operation 'contract' causes the rewriting of a pair of Levi-Civita tensors into sums and products of Kronecker delta's. After the indices m1, m2, m3 have been summed over we have the proper determinant left as is shown in the printout. The contract statement contracts one pair of Levi-Civita tensors only in each term (unless there are none of course). The pair that is contracted is the pair that results in the fewest number of terms after the contraction. In other words: the most complicated contraction is kept for the last. This results in the greatest speed. If there is more than one pair of Levi-Civita tensors there is no need to work with several consecutive 'contract' statements. If the contract statement is followed by a number FORM keeps contracting pairs of Levi-Civita tensors until there are the specified number of tensors left (or one more of course). So the statement 'contract 0' will contract all Levi-Civita tensors. There are more options but for them the user should consult the chapter on Levi-Civita tensors.

There is also a directive which we hadn't used thus far. This is the '.sort' directive. It tells FORM that this is the end of the current module and that we want it executed. We are not terminating the program however and we want to continue with the current expression(s). What it means basically is that FORM should execute all the specified statements and sort the resulting terms so that the output is in a unique form. After that we may continue with the next module. In the current program the reason that we used the '.sort' was that we wanted to show that the trick with the Levi-Civita tensors did indeed give the determinant of the 3 by 3 matrix M.

Next we want to tell FORM what the components of M are in terms of the coefficients of the polynomial that we started with.

This is done with the 'id' statement. 'id' stands for identify and the 'id' statement is the most important statement in symbolic manipulation. It allows the user to compose complicated operations by successive substitutions. In the current program we don't want to do anything fancy. We just want to replace the matrix elements by their 'values'. Having done so, FORM reworks the output and gives the final result.

It would be too ambitious to show the full syntax of the 'id' statement right away. Because it is so important it has several options and there can be many 'patterns' on the left hand side of the equals sign. We define a pattern as a left hand side of an id-statement. FORM should try to match patterns with the contents of each of the individual terms of the current expression. There is one strict rule: patterns may contain only one term and their coefficients must be equal to one.

The above example shows also how commentary is included in a program. Each line that starts with the character * in column 1 is considered to be commentary. This commentary is allowed to be between the various lines that make up a complicated expression.

In the next example we will look at the differentiation of a product of noncommuting functions. This will involve a new technique: the use of wildcards. Wildcards are generic objects that are used in patterns and that can then match a class of objects.

```
Functions g1,g2,g3,g4,dx,g;
Symbols x,n;
Local F = g1(0,x)*g2(0,x)*g3(0,x)*g4(0,x);
Multiply,left,dx;
repeat;
    id  dx*g?(n?,x) =
            g(n+1,x) + g(n,x)*dx;
endrepeat;
id  dx = 0;
```

```
    print;
    .end

Time =     0.20 sec     Generated terms =          8
              F         Terms left      =          4
                        Bytes used      =        514

   F =
      g1(0,x)*g2(0,x)*g3(0,x)*g4(1,x) + g1(0,x)*
      g2(0,x)*g3(1,x)*g4(0,x) + g1(0,x)*g2(1,x)*
      g3(0,x)*g4(0,x) + g1(1,x)*g2(0,x)*g3(0,x)*
      g4(0,x);
```

We have chosen a notation in which the first argument of our functions indicates the level of the derivative. so `g2(3,x)` would indicate the third derivative of g2. The multiply statement has three varieties: 'Multiply,left', 'Multiply,right' and just 'Multiply'. In the last case FORM will figure out what is fastest. In our case we want to work the dx through from the left to the right so we multiply the expression at the left side by dx. Any legal expression could be used in the multiply statement.

The next new statement is the repeat statement. It has to be matched by an endrepeat statement and its working is that the block of statements in between are executed as a block until they cause no more action. The main restriction is that all statements inside the repeat/endrepeat including the repeat and endrepeat statements themselves must be inside the same module. In the current program this means that the id-statement is executed until it has no effect any more.

This leaves us with the id-statement itself. It contains two so called wildcards: `g?` and `n?`. The first one means 'any function' because g is a function. The second one stands for 'any symbol, number or argument with a scalar nature'. This means that when

dx is left of the function g1 g will match the g1 and in the right hand side of the id-statement g will be replaced by g1. n will match the zero and so in the right hand side n will be replaced by 0. If there would be another number or a symbol or an expression n would take the identity of that object as is shown in the next example:

```
    Functions g1,g2,dx,g;
    Symbols x,n,m;
    Local F = g1(0,x)*g2(m,x);
    Multiply,left,dx*dx*dx;
    repeat;
        id  dx*g?(n?,x) =
                g(n+1,x) + g(n,x)*dx;
    endrepeat;
    id  dx = 0;
    print;
    .end
```

```
Time =     0.58 sec     Generated terms =         82
              F         Terms left      =          4
                        Bytes used      =        370

   F =
   g1(0,x)*g2(3 + m,x) + g1(3,x)*g2(m,x) + 3*g1(2,x)
   *g2(1 + m,x) + 3*g1(1,x)*g2(2 + m,x);
```

The integration of polynomials becomes also easy when wildcards are used:

```
    Symbols x,j,dx,n;
    Local F = sum_(j,0,10,x^j);
    multiply,dx;
    id  dx*x^n? = x^(n+1)/(n+1);
```

```
    format 56;
    print;
    .end

Time =     0.25 sec     Generated terms =         53
              F         Terms left      =         11
                        Bytes used      =        178

   F =
   1/11*x^11 + 1/10*x^10 + 1/9*x^9 + 1/8*x^8 + 1/7*
   x^7 + 1/6*x^6 + 1/5*x^5 + 1/4*x^4 + 1/3*x^3 + 1/2
   *x^2 + x;
```

To generate a rather lengthy polynomial we have used the `sum_` function which has either 4 or 5 arguments. The first argument is the summation parameter, the second the first value, the third the last value. If there are 5 arguments the fourth argument is the increment. If there are four arguments the increment is taken to be one. The last argument is the object to be summed. By now the id-statement should be clear: we take x to any power and n takes the value of that power. Can the reader tell what went wrong in the next program?

```
    Symbols x,j,dx,n;
    Local F = sum_(j,-5,5,x^j);
    multiply,dx;
    id  dx*x^n? = x^(n+1)/(n+1);
    print;
    .end
Division by zero during normalization
```

We see here an example of a run time error. Errors can occur in many different ways and not all ways are caught properly by FORM. Usually it will sense an error condition and give the user

some idea where to look. This is particularly the case when it turns out that one of its buffers is not large enough. There are several types of errors: errors in the syntax and errors during execution. The errors in the syntax can be directly fatal or they can allow FORM to proceed with checking the rest of the syntax. The execution time errors can be rather soft, in which case they result in a warning only and execution continues, or as in the above case execution had to be halted because any answer would have been incorrect.

For the reader to try:

1: There is a different way to generate determinants with vectors rather than with a function as was done in the first example. When one uses the products of `e_(p1,p2,p3)` and `e_(q1,q2,q3)` the various dotproducts that occur after the contraction are equivalent to the matrix elements. Write a program that solves the first problem this way.

2: Use the repeat statement to generate the 9-th and the 10-th Fibonacci numbers. Why is this method not really very efficient?

3: Repair the last example in this section (ie make it give the proper integral).

2.4 Some more wildcards

In the previous examples we saw some rather simple uses of wildcards. There are however much more complicated possibilities, supporting the thesis that the id-statement is the most important statement in symbolic manipulation. In principle each occurrence of a symbol, index, vector or function can be wildcarded unless they are part of a composite function argument. We will see some examples:

```
Vectors,p,q,r,s;
Indices,mu,nu,rho;
CFunction,g;
Nwrite statistics;
Local F = p(mu)*q.r*p(rho)
        + q(mu)*p.r*q(rho)
        + r(mu)*p.q*r(rho);
id,p(mu?) = s(mu);
print;
.sort

F =
   q(mu)*q(rho)*p.r + r(mu)*r(rho)*p.q + s(mu)
   *s(rho)*q.r;

id,p?(rho) = p.p*p(rho);
print;
.sort

F =
   q(mu)*q(rho)*p.r*q.q + r(mu)*r(rho)*p.q*r.r
    + s(mu)*s(rho)*q.r*s.s;

id,p?.r = p.p*p.r;
```

```
 print;
 .sort

F =
   q(mu)*q(rho)*p.p*p.r*q.q + r(mu)*r(rho)*p.q
   *r.r^2 + s(mu)*s(rho)*q.q*q.r*s.s;

 id,q?.q? = 1;
 print;
 .sort

F =
   q(mu)*q(rho)*p.r + r(mu)*r(rho)*p.q + s(mu)
   *s(rho)*q.r;

 id,p?.q? = g(p,q);
 print;
 .sort

F =
   g(p,q)*r(mu)*r(rho) + g(p,r)*q(mu)*q(rho)
    + g(q,r)*s(mu)*s(rho);

 id,p?(mu)*p?(nu?) = g(p,mu,nu);
 print;
 .end

F =
   g(p,q)*g(r,mu,rho) + g(p,r)*g(q,mu,rho) +
   g(q,r)*g(s,mu,rho);
```

The above is an exercise in wildcarding modes of vectors with indices and dotproducts. In addition there is the 'nwrite statistics'

statement which turns off the writing of the run time statistics. This allows us to concentrate our attention to the interplay between wildcards and output. When a wildcard occurs twice in the left hand side as in q?.q? both occurrences have to match the same object. Therefore q?.q? will match p.p but it won't match p.q as can be seen in the output. It is of course also possible to have wildcard powers of dotproducts containing wildcard vectors. In that case there is no match when the power would become zero. This should be rather understandable as there is no vector to substitute for the wildcard vector and FORM cannot construct a proper right hand side if this contains this vector. When dotproducts don't contain wildcard vectors their wildcard power can indeed become zero.

With functions things become more complicated:

```
 Symbols,a,b,c;
 CFunctions,f,g,h;
 Nwrite statistics;
 Local expr =
     f(a,b)+g(b,c)+h(c,a);
 id,f?(b,c)=2*f(a,b);
 print;
 .sort

expr =
   f(a,b) + 2*g(a,b) + h(c,a);

 id,f(c?,b) = c*f(c+1,b);
 print;
 .sort

expr =
   f(1 + a,b)*a + 2*g(a,b) + h(c,a);
```

```
 id,f(c?,b) = c*f(c+1,b);
 print;
 .end

expr =
   f(2 + a,b)*a^2 + f(2 + a,b)*a + 2*g(a,b) +
   h(c,a);
```

The first wildcard substitution involved the function itself, so it would only match a function with the given fixed argument field. The second substitution had 'c' match the first argument of f if the second would be 'b'. When a symbol is used as a wildcard argument it will match a full argument if this argument isn't an index or a vector (or vectorlike). This is exemplified in the last substitution. This property can be used profitably when expanding compactified notations:

```
 Symbols,n,x;
 CFunctions,Pochhammer;
 Nwrite statistics;
 Local expr =
     Pochhammer(x,4);
 repeat;
     id,Pochhammer(x?,0) = 1;
     id,Pochhammer(x?,n?)=x*Pochhammer(x+1,n-1);
 endrepeat;
 print;
 .end

expr =
   x^4 + 6*x^3 + 11*x^2 + 6*x;
```

It should not be difficult to use the above example to figure out what the definition of the Pochhammer function is.

If x? (with x a symbol) can pick up any argument that isn't an index or vectorlike, how then do we pickup these other arguments? The answer should be clear: with indices and vectors. There are some subtleties though:

```
 Indices,mu,nu,alfa,beta;
 Vectors,p,q,r,s;
 CFunction,f;
 Nwrite statistics;
 Local,expr =
     f(mu,p+q);
 id,f(mu,r?) = f(r,mu);
 print;
 .sort

expr =
   f(p + q,mu);

 id,f(nu?,mu) = f(mu,nu);
 print;
 .end

expr =
   f(mu,p) + f(mu,q);
```

In this example we see that in the first substitution r replaces the vectorlike argument p+q. In the second substitution we have an index wildcard. This indicates to FORM that if a vector is found in this position its occurrence is due to the compactified notation for summed over indices. Therefore there must have been an index at this position and the composite vector was outside the function. So the function must be linear in this argument and this property is used. Therefore we have two terms in the output

rather than `f(mu,p+q)`. If the user doesn't want this linearization he should use a wildcard vector.

At times it is very useful to have a single wildcard for a group of function arguments, regardless the number of arguments that is involved. Such a wildcarding is also available. There are 10 wildcard variables that can serve as such. On the left hand side they are indicated with one to 10 question marks. On the right hand side they are used with one to 10 dots in succession. Three dots match the variable with three question marks. Below is a very simple example:

```
Symbols,a,b,c;
CFunctions,f,g,h;
nwrite statistics;
Local,expr =
    f(a)+f(a,b)+f(a,b,c);
id  f(??) = g(..)+h(..);
format 50;
print;
.end

expr =
   g(a) + g(a,b) + g(a,b,c) + h(a) + h(a,b) +
   h(a,b,c);
```

We have used the variable with the two question marks, so on the right hand side we need the variable with the two dots. We can see here clearly that any argument field has been good for a match, independent of the number of arguments. It is of course possible to make much more meaningful examples:

```
Indices,m1,m2,m3,m4,m5,m6,m7,m8;
Symbols,a1,a2,a3,a4,a5,a6,a7,a8;
CFunction,M;
```

```
nwrite statistics;
Local expr =
    + M(m1,m2,a1) * M(m3,m4,a3)
    * M(m5,m6,a5) * M(m7,m8,a7)
    * M(m2,m3,a2) * M(m4,m5,a4)
    * M(m6,m7,a6) * M(m8,m1,a8);
repeat;
    id,M(m1,m2?,??)*M(m2?,m3?,???)
        = M(m1,m3,..,...);
endrepeat;
format 50;
print;
.end

expr =
   M(m1,m1,a1,a2,a3,a4,a5,a6,a7,a8);
```

We have used M as a matrix of which the first two arguments indicate the row and the column, and the rest of the arguments indicate dependencies. Assuming that we are summing over all the indices we want to write the whole expression as a single trace. Now this argument field wildcarding comes in very handy. Any product that has the proper common index is strung together, independent of what the other arguments are, and independent of how many there are. There are some limitations of course: There may be only one argument field wildcard per occurrence of a function in the left hand side, because if there were more of them inside a single function the matching becomes at the best inefficient and at the worst ambiguous. This kind of wildcarding can be used for nearly any type of function. Only the built in functions d_, e_ and g_ are excluded from this rule. d_ is the Kronecker delta, e_ is the Levi-Civita tensor and g_ is the gamma matrix (see later in this manual).

There is one type of wildcarding which is currently forbidden: patterns like `f(a+x?)` are not allowed. Allowing wildcarding inside composite function arguments can become so complicated that no attempt has been made to implement it thus far. This rule holds also for elements of composite denominators and of exponent functions that cannot be evaluated immediately, due to a noninteger power.

There is one more simple pattern which can also be wildcarded. If we substitute just a vector, all the occurrences of this vector inside vectors-with-arguments, dotproducts and the special functions e_ (Levi-Civita tensor) and g_ (gamma matrices) are replaced:

```
Vectors,p,q,r,s;
Indices,mu,nu;
CFunction,h;
nwrite statistics;
Local expr =
    p(mu)*p.q+e_(p,q,mu)+h(mu,p);
id  p = (r+s)/2;
print;
.end

expr =
   1/4*r(mu)*q.r + 1/4*r(mu)*q.s + 1/4*s(mu)*
   q.r + 1/4*s(mu)*q.s - 1/2*e_(q,r,mu) - 1/2*
   e_(q,s,mu) + h(mu,p);
```

The example shows that all p's were substituted except for the one inside the argument of the function h. It is also possible to use p? in the left hand side of the id-statement. This is rarely useful though.

The last example is more for fun. We see here a new type of instruction though. When an instruction starts with the character # it is meant for the preprocessor. The preprocessor deals with the input before the actual algebraic part of FORM gets to see it. We will see more examples of preprocessor instructions in later sections. Here we see the do-loop: a capacity that allows us to repeat a given number of input lines. These lines may include any type of statements, including end-of-module statements. The user should try the following example by himself:

```
CFunction o;
Local O = o;
#do i = 1,9
    id,o(??) = o(o(..),..);
#enddo
print;
.end
```

Before running it one should make a file named 'form.set' with the following lines in it:

```
LargeSize 0
ScratchSize 17000
MaxTermSize 17000
WorkSpace 130000
```

The keywords in this file should start in column 1. This file allows FORM to choose a different layout of its buffer space that is more suitable for this problem. With these parameters one can have the do-loop in the program deal with up to 10 repetitions. The result is an example of a fractal in the shape of a formula.

2.5 Sets and wildcarding

The wildcarding we have seen thus far involved wildcards that would match any variable of the right type. This is often more than that we want. To avoid such general nature there are two ways. 1: write many statements without wildcards, so that only those objects that we want to replace are indeed replaced. 2: Introduce a new type of variables that are called 'sets' which contain elements with which the wildcard is allowed to match. This second way is clearly superior as long as we can introduce an elegant notation for it.

```
Symbols   a,b,c,d,e,f;
Set       half:a,c,e;
CFunction H,M;
nwrite statistics;
Local     expr =
        H(a)+H(b)+H(c)+H(d)+H(e)+H(f);
id,  H(a?half) = M(a);
print;
.end

expr =
   H(b) + H(d) + H(f) + M(a) + M(c) + M(e);
```

Each set has to be declared and per statement one can only declare one set. The set has a name which should be followed by a colon after which there follow the elements. All elements must be of the same type. For the type of the elements commuting functions and noncommuting functions are considered to have the same type. The commutational properties are considered to be secondary variations. In the id-statement the name of the set is appended to the question mark. This means that a?half must be an element of the set 'half'.

How great a help these sets can be becomes clear once we try to integrate some complicated expressions.

```
nwrite statistics;
Symbols   x,n,y,[log(x)],dx;
Set  fromx:x,[log(x)];
Local expr =
        (x+[log(x)])^2;
*
*   Now we want to integrate:
*
multiply,dx;
id,select,fromx,dx*x^n? = x^(n+1)/(n+1);
id,select,fromx,dx*x^n?*[log(x)] =
    x^(n+1)/(n+1)*([log(x)]-1/(n+1));
id,select,fromx,dx*x^n?*[log(x)]^2 =
    x^(n+1)/(n+1)*([log(x)]^2-2*[log(x)]/(n+1)
    +2/(n+1)^2);
print;
.end

expr =
   1/3*x^3 + x^2*[log(x)] - 1/2*x^2 + x*
   [log(x)]^2 - 2*x*[log(x)] + 2*x;
```

The select option of the id-statement shows us several things. 1: id-statements can have options. 2: They are inserted between the id and the left hand side. The 'select' should be followed by the names of one or more sets. The substitution will be made only if no elements of the sets that were mentioned are left as individual elements. This last thing means that individual powers of symbols cannot be left, but if the symbol is an argument of a function it can be left. This restriction may be lifted in future versions. In

the current example it forced us to write the log(x) as a symbol, using the notation with the straight braces.

The advantage of the select option is here that we don't have to worry so much about the order of the statements. Had we not used this option, then the answer would have been different:

```
nwrite statistics;
Symbols   x,n,y,[log(x)],dx;
Set  fromx:x,[log(x)];
Local expr =
        (x+[log(x)])^2;
multiply,dx;
*
*   Don't use 'select'
*
id,dx*x^n? = x^(n+1)/(n+1);
id,dx*x^n?*[log(x)] =
    x^(n+1)/(n+1)*([log(x)]-1/(n+1));
id,dx*x^n?*[log(x)]^2 =
    x^(n+1)/(n+1)*([log(x)]^2-2*[log(x)]/(n+1)
    +2/(n+1)^2);
print;
.end

expr =
   1/3*x^3 + x^2*[log(x)] + x*[log(x)]^2;
```

This answer is wrong. In this case we would have to be meticulous about the order of the statements:

```
nwrite statistics;
Symbols   x,n,y,[log(x)],dx;
Set  fromx:x,[log(x)];
Local expr =
```

```
        (x+[log(x)])^2;
multiply,dx;
*
*   Don't use 'select'
*
id,dx*x^n?*[log(x)]^2 =
    x^(n+1)/(n+1)*([log(x)]^2-2*[log(x)]/(n+1)
    +2/(n+1)^2);
id,dx*x^n?*[log(x)] =
    x^(n+1)/(n+1)*([log(x)]-1/(n+1));
id,dx*x^n? = x^(n+1)/(n+1);
print;
.end

expr =
   1/3*x^3 + x^2*[log(x)] - 1/2*x^2 + x*
   [log(x)]^2 - 2*x*[log(x)] + 2*x;
```

When the integrations become more complicated it may become very difficult to keep track of the proper order of the statements, hence the 'select' option.

During the wildcarding sets can also be used a little bit as arrays:

```
nwrite statistics;
Symbols a1,a2,a3,b1,b2,b3,x,n;
Set     aa:a1,a2,a3;
Set     bb:b1,b2,b3;
CFunctions g1,g2,g3,g;
Set     gg:g1,g2,g3;
Local expr =
    g(a1)+g(a2)+g(a3)+g(x);
id,g(x?aa[n]) = gg[n](bb[n]);
```

```
 print;
 .end

expr =
   g1(b1) + g2(b2) + g3(b3) + g(x);
```

The notation is here that x must belong to the set aa and that n becomes the number of the element in the set to which x matched. This number can then be used in the right hand side as if it is a normal symbol or to indicate an element of a set. No arithmatic can be done to compute other elements of a set as would be the case in `bb[n+1]`. If the user would like such a thing he should define another set of which the elements are shifted by one.

Array elements can also be used with a numerical argument in the input. In that case one should realize that the counting of array elements starts at one:

```
 nwrite statistics;
 Symbols a1,a2,a3;
 Set     aa:a1,a2,a3;
 CFunction g;
 Local expr =
     g(aa[1])+g(aa[2])+g(aa[3]);
 print;
 .end

expr =
   g(a1) + g(a2) + g(a3);
```

This can be of importance when we use preprocessor variables like the do-loop parameter to indicate set elements. More about this later.

There is another way to map the elements of one set onto those of another set:

```
nwrite statistics;
Symbols a1,a2,a3,b1,b2,b3,x;
Set     aa:a1,a2,a3;
Set     bb:b1,b2,b3;
CFunction g;
Local expr =
    g(a1)+g(a2)+g(a3);
id  g(x?aa?bb) = g(x);
print;
.end

expr =
   g(b1) + g(b2) + g(b3);
```

The wildcarding x?aa?bb means that x must belong to the set 'aa' and in the right hand side each occurrence of x will be replaced by the corresponding element of 'bb'. The sets 'aa' and 'bb' must have the same number of elements or there will be an error message.

When the array elements are used during wildcarding the array element should be indicated by a symbol. This symbol can also be used for more wildcarding action in the same statement:

```
nwrite statistics;
Symbols a1,a2,a3,b1,b2,b3,x,y,n;
Set     aa:a1,a2,a3;
Set     bb:b1,b2,b3;
CFunction g,h;
Local expr =
    h(a1)*b1^3+h(a2)*b2^2+h(a3)*b3;
id  h(x?aa[n])*y?bb[n]^n? = g(x,y)*n;
```

```
print;
.end

expr =
   g(a1,b1)*b1^2 + 2*g(a2,b2) + h(a3)*b3;
```

In the first term n becomes 1 and so only one power of b1 can be absorbed. In the third term there aren't enough powers of b3 to allow a match. Note also that x and y have kept their traditional identity, even though we could have obtained their contents as the array elements `aa[n]` and `bb[n]`.

Finally we'll have a look at a more realistic example of the differentiation of a class of functions of a single symbolic variable:

```
nwrite statistics;
Symbol  x,y,n;
CFunctions sin,cos,exp,g;
Functions [sin],[cos],[-sin],[exp],f,dx;
Set   commuting:sin,cos,exp;
Set   noncommuting:[sin],[cos],[exp];
Set   derivative:[cos],[-sin],[exp];
Local expr =
    x^3*(sin(x)+cos(x)*exp(x));
*
*   Make noncommuting functions.
*   The order is unimportant.
*
id  g?commuting?noncommuting(x) = g(x);
multiply,left,dx;
*
*   Now the derivative of the functions
*
repeat;
    id,dx*g?noncommuting[n](x) =
```

```
            derivative[n](x)+g(x)*dx;
        id,[-sin](x) = -[sin](x);
    endrepeat;
    *
    *   The derivative of the polynomial part
    *
    id  dx*x^n? = n*x^(n-1);
    *
    *   Back to the regular functions:
    *
    id  f?noncommuting?commuting(x) = f(x);
    print;
    .end

  expr =
     3*sin(x)*x^2 - sin(x)*exp(x)*x^3 + cos(x)*
     x^3 + cos(x)*exp(x)*x^3 + 3*cos(x)*exp(x)*
     x^2;
```

Procedures of this kind can be made once for a class of problems and then stored in a library file. These library files are managed by the preprocessor, so we will see more about them in the section on the preprocessor.

2.6 Manipulations with expressions

From now on we have modified the environment slightly by means of a setup file. The default is now not to print the statistics.

Now we have seen some of the basics operations it is time to study the expressions in more detail. There are several types of expressions. We saw already the local expressions. The other type consists of global expressions and those come again in two varieties: the active global expressions and the stored global expressions. Global expressions serve two purposes: they can be used with a parameter field and they can be stored and saved for future programs. Let us make some simple examples:

```
 Symbols a,b;
 Global F(a,b) = (a+b)^4;
 print;
 .store

F(a,b) =
   a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4;

 Symbols c,d;
 Local G = F(c,d);
 print;
 .end

G =
   c^4 + 4*c^3*d + 6*c^2*d^2 + 4*c*d^3 + d^4;
```

The .store indicates an end of the module. The formula for F is worked out and then, before continuing with the next module FORM does some cleaning up. All local expressions are removed from the system. The global expressions are stored away in the

storage file. Then all declarations made undone. To this last action there is an exception as we will see pretty soon. In our example F is put in the storage file. From now on F cannot be modified any more. It can be used however in the definition of new expressions. When this is done the type of the parameters must match the types in the definition of F exactly. Even the replacement of either c or d by a number is not allowed. This should not be a serious limitation as a symbol can be replaced rather easily by a number or any other expression in an id-statement.

The use of F isn't restricted to the definition of new expressions. F can also be used in the right hand side of id-statements or at any other place were formulae like those right hand sides can be used. FORM doesn't know about the contents of an expression, so when powers of expressions are used FORM will never use the binomial expansion. There could be noncommuting objects after all.

Saving the results of a computation for a future run is done with the 'save' statement. Its syntax is rather simple. It needs a file name and possibly one or more names of stored global expressions. If no names are mentioned all stored expressions are saved into the file with the given file name:

```
Symbols a,b;
Global F(a,b) = (a+b)^4;
print;
.store

F(a,b) =
   a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4;

save simple.sav;
.end
```

In the above case the file 'simple.sav' is created and the stored

expression F is put in it. If there were already a file by the name 'simple.sav' its contents would be lost.

We can use the results of the above computation with the 'load' statement. This statement has the same syntax as the 'save' statement. When an expression is loaded it is put in the storage file and it gains the same status as an expression that would have been stored in the current program:

```
    Load simple.sav;
 F loaded
    Symbols c,d;
    Local G = F(c,d);
    print;
    .end

   G =
      c^4 + 4*c^3*d + 6*c^2*d^2 + 4*c*d^3 + d^4;
```

Note that FORM tells which expressions were loaded. This gives the user some idea of what other expressions reside in the same file. In the future there will be an 'audit' command to give a precise layout of the expressions in a file, and a list of all variables that are used in them.

Global files are stored together with a list of all there variables that are actually used in them. When the expression is used as part of a new expression these namelists are also read and compared with the currently active namelists. If there is a conflict there may be either a warning or an error message, depending on how serious the conflict is. If a variable doesn't exist in the current name tables it will be added by FORM and after the next .sort the user can use these variables as if he had declared them himself. It is safer though to not rely on this mechanism. Using hidden properties can cause surprises and sometimes lead to quite unexpected errors.

On the other hand FORM removes all variables from the namelists after a .store statement, so this would lead to many repetitions of declarations. To avoid this there is the .global instruction. This instruction is used to construct a module that contains only declarations. All declarations that are active at the moment of a .global instruction are kept over a .store instruction:

```
 Symbols a,b,c,d;
 .global
 Global F = (a+b)^2;
 print;
 .store

F =
   a^2 + 2*a*b + b^2;

 Local G = F;
 Local H = (c-d)^3;
 print;
 .end

G =
   a^2 + 2*a*b + b^2;

H =
   c^3 - 3*c^2*d + 3*c*d^2 - d^3;
```

Here we used F without parameters. In the second part of the program the properties of c and d were still known, because all symbols that we used were declared globally at the beginning.

When working on a large project it is often advisable to put all declarations in a separate file:

```
 Symbols a,b,c,d;
```

```
Vectors p,q,r,s;
Functions f1,f2,f3;
CFunctions g1,g2,g3;
```

We have put these declarations here in the file 'declare.h' and use it with the #include instruction:

```
#include declare.h
Local F = (a+b)^2;
print;
.end
```

The # indicates that the preprocessor should get busy. It opens the file and treats its contents as if they were in the regular input file. The output of the program will look like:

```
#include declare.h
Symbols a,b,c,d;
Vectors p,q,r,s;
Functions f1,f2,f3;
CFunctions g1,g2,g3;

Local F = (a+b)^2;
print;
.end

F =
   a^2 + 2*a*b + b^2;
```

This allows for a rather convenient continuity. It can also be that the user doesn't want to see the contents of an 'include file'. To mask a part of the input for the listing mechanism one may insert the instructions `#-` for turning the listing off and `#+` for turning the listing on. We can give the file 'declare.h' now the contents:

```
#-
Symbols a,b,c,d;
Vectors p,q,r,s;
Functions f1,f2,f3;
CFunctions g1,g2,g3;
#+
```

after which the job produces the following output:

```
#include declare.h
#-

Local F = (a+b)^2;
print;
.end

F =
   a^2 + 2*a*b + b^2;
```

The listing of `#-` warns the user that part of the input is not shown in the output.

The above shows what can be done with expressions in a global sense. There are however many more things that can be done locally. The now following holds both for local and for global expressions. The only difference between local and global expressions lies in what happens with them when a .store instruction is executed. Also local expressions cannot have parameters, but neither can the parameters of a global expression be used before the expression has been stored.

Once an expression has been defined it can be used in the right hand side of another expression or an id-statement. This is preferably done after a .sort instruction but this is not necessary. The difference is merely a matter of economy. When it is done

before a .sort instruction the occurence of the expression in the right hand side is replaced by the definition of the expression. So:

```
    Symbols a,b,c;
    Local F = (a+b+c)^4-a*(a+b+c)^3
     - b*(a+b+c)^3;
    Local G = F * F;
    write statistics;
    .end

Time =       0.15 sec    Generated terms =         35
                F        Terms left      =         10
                         Bytes used      =        210

Time =       4.75 sec    Generated terms =       1225
                G        Terms left      =         28
                         Bytes used      =        618
```

is much more time consuming than needed, because the computation of G is performed before the cancellations inside F have taken place. The proper way is:

```
    Symbols a,b,c;
    Local F = (a+b+c)^4-a*(a+b+c)^3
     - b*(a+b+c)^3;
    write statistics;
    .sort

Time =       0.14 sec    Generated terms =         35
                F        Terms left      =         10
                         Bytes used      =        210
    Local G = F * F;
    .end
```

```
Time =       0.18 sec    Generated terms =         10
                F        Terms left      =         10
                         Bytes used      =        210

Time =       0.42 sec    Generated terms =        100
                G        Terms left      =         28
                         Bytes used      =        618
```

Occasionally we use an expression, like F in the above example for an intermediate result only. We don't need it any more after some steps. It would be wasteful to keep carrying it around, possibly making all kinds of substitutions on it, because we make substitutions on the other expressions. One way out would be to declare the expression(s) that used F to be global expressions and then issue a .store instruction. This isn't very elegant. The better way is to used the 'drop' statement. When this statement is used the expression can still be used in the current module, but after the next .sort (or .store) instruction the expression doesn't exist any more. Its name is available again. In the above job we would use it as follows:

```
    Symbols a,b,c,d,e;
    Local F = (a+b+c)^4-a*(a+b+c)^3
     - b*(a+b+c)^3;
    write statistics;
    .sort

Time =       0.18 sec    Generated terms =         35
                F        Terms left      =         10
                         Bytes used      =        210
    Drop F;
    Local G = F * F;
    .end
```

```
Time =       0.44 sec    Generated terms =        100
                G        Terms left      =         28
                         Bytes used      =        618
```

Note that F doesn't appear in the final statistics any more. It cannot be acted upon any more in the second module, because it is dropped. It is allowed to mention more than one expression in a single drop statement. What also occurs frequently is the simultaneous treatment of several expressions. At some point one would like to make some substitutions in some of the expressions, but not in others. One can inactivate an expression for the range of a module with the 'skip' statement. The expression is still available for the use in a right hand side, but no operations are performed on it in the current module:

```
Symbols a,b,c,d,e;
Local F = (a+b)^3;
Local G = (a+b)^4;
id b = c+d;
.sort
skip F;
id d = b - c;
.sort
id a = e;
print;
.end
```

```
F =
   c^3 + 3*c^2*d + 3*c^2*e + 3*c*d^2 + 6*c*d*e + 3
   *c*e^2 + d^3 + 3*d^2*e + 3*d*e^2 + e^3;

G =
   b^4 + 4*b^3*e + 6*b^2*e^2 + 4*b*e^3 + e^4;
```

In this case the id-command 'd = b - c' was executed only in the expression G. This is rather evident in the output. Also the 'skip' statement can have more than one expression in its parameter field.

The final property of expressions is slighty more involved but it can be very useful. It has to do with the control over the output format of an expression. Often the display of the output becomes much clearer when there are some variables taken outside parentheses. This can be obtained with the bracket statement:

```
Symbols a,b,c,d,e;
Local F = (a+b+c)^6;
Bracket a;
print;
.end

F =
    + a^6 * ( 1 )
    + a^5 * ( 6*b + 6*c )
    + a^4 * ( 15*b^2 + 30*b*c + 15*c^2 )
    + a^3 * ( 20*b^3 + 60*b^2*c + 60*b*c^2 + 20*
      c^3 )
    + a^2 * ( 15*b^4 + 60*b^3*c + 90*b^2*c^2 + 60*
      b*c^3 + 15*c^4 )
    + a * ( 6*b^5 + 30*b^4*c + 60*b^3*c^2 + 60*b^2
      *c^3 + 30*b*c^4 + 6*c^5 )
    + b^6 + 6*b^5*c + 15*b^4*c^2 + 20*b^3*c^3 + 15
      *b^2*c^4 + 6*b*c^5 + c^6;
```

It is possible to mention more variables in a single bracket statement. All occurrences of all variables in the bracket statement are taken outside parentheses. There can however be only one bracket statement per module. If there is more than one, only the last one is relevant. The bracket statement is only active

for the module in which it is defined and influences not only the screen representation, but also the way in which the expression is kept over the .sort or the .store instruction. This enables the user to refer to the contents of the brackets:

```
    Symbols a,b,c,d,e;
    Local F = (a+b+c)^6;
    Bracket a;
    .sort
    drop F;
    write statistics;
    Local G = 8*F[a^2]^2 - 15*F[a]*F[a^3];
    Local H = F[1];
    print;
    .end

Time =       0.41 sec    Generated terms =         49
               G         Terms left      =          0
                         Bytes used      =          2

Time =       0.43 sec    Generated terms =          7
               H         Terms left      =          7
                         Bytes used      =        134

   G = 0;

   H =
      b^6 + 6*b^5*c + 15*b^4*c^2 + 20*b^3*c^3 + 15*
      b^2*c^4 + 6*b*c^5 + c^6;
```

One refers to the contents of a bracket in the expression F by placing the part that is outside the brackets enclosed in square braces after the name F. The part that has nothing outside brackets is refered to as `F[1]`. This feature can be very useful when

solving equations. This is shown in the next example in which we have two linear equations in three variables. We make these into one equation in two variables:

```
Symbols a,b,c,x,y,z;
Local E1 = (a+b)^2*x+(b+c)^2*y+(c+a)^2*z;
Local E2 = (a-b)^2*x+(b-c)^2*y+(c-a)^2*z;
Bracket z;
.sort
drop E1,E2;
Local F1 = E1*E2[z] - E2*E1[z];
Bracket x,y;
print;
.end

F1 =
    + x * ( 4*a^3*b - 4*a^3*c - 4*a*b^2*c + 4*a*b*
      c^2 )
    + y * ( 4*a^2*b*c - 4*a*b^2*c - 4*a*c^3 + 4*b*
      c^3 );
```

We have multiplied the first equation (we assume that E1 and E2 are equal to zero) by the coefficient of z in the second equation and vise versa. The difference doesn't contain z any more.

When a bracket statement is used before a .store instruction the global expressions are put in the storage file with the bracketing active. If such an expression is saved one can use the bracketing even in a later program.

2.7 The superstructure of the preprocessor

The preprocessor of FORM is a program unit that manipulates the input before it is offered to the compilation and execution units. Anything that is done by the preprocessor has therefore no direct influence on the terms of an expression. It is merely there to allow the user to write his programs in a more concise and less laborious way. As such it is a rather autonomous unit with its own calculator, its own if instruction that is entirely different from the if statement that works on the level of terms (which we will encounter later). There are two types of preprocessor commands. Of the first type we know nearly everything already. These are the commands that start with a period. They are .end, .sort, .store, .global and .clear. The last command is for very special use only so the user is refered to the more detailed chapter on the preprocessor if he wants to know more about it. It was built in merely to have a convenient way to make sure that all examples of the manual will run flawlessly. The other preprocessor commands start with a hash mark (#) and are refered to as instructions. Finally there are the comment lines. They are filtered out by the preprocessor after printing them in the output (unless the listing of the input has been turned off). In this section we will study the # instructions.

Before we start to look at any examples we should get acquainted with the preprocessor variables. Their names are completely independent of the names of the other variables in the program. One can therefore have the regular variable 'i' and the preprocessor variable 'i'. The difference lies in the use. When a preprocessor variable is defined it is refered to plainly by the characters of its name but when it is used its name is enclosed between single quotes. This last convention makes it possible to concatenate regular strings of characters and preprocessor variables to

form larger strings of characters:

```
Symbols a1,a2,a3,a4,a5;
Local F =
#do i = 1,5
    + a'i'^''i''i'
#enddo
    ;
print;
.end

F =
   a1^11 + a2^22 + a3^33 + a4^44 + a5^55;
```

The above is a good example of that the preprocessor manipulates the input only. It generated five lines which are then seen by the rest of FORM as the body of the right hand side of the defining statement of F. The method of having a loop parameter compose names of variables is very useful and can make many a long input much shorter. It becomes even more powerful with the mini calculator which allows for rather primitive computations. The object `{'i'+1}` is a string that can be used in the same way as 'i' but its interpretation as a number is one larger than 'i'. When these curly brackets are used the preprocessor variables inside are interpreted as (short) integer numbers, the necessary computations are performed and the result is transformed back into a string. The calculator knows parentheses, addition, subtraction, multiplication, division (integer division with rounding toward zero) and modulus (a`%`b is the modulus of a with respect to b). If the string between the curly brackets contains any illegal characters after the preprocessor variables have been inserted –legal are only 0-9, +, -, `*`, /, `%`, (and)– an error message will be issued and execution will be halted immediately. There will be no further checking of the input for errors. This holds for most of

the errors against the preprocessor syntax, because those errors can really mess up the input.

```
    Local F0 = 1;
    Local F1 = 1;
    #do i = 2,10
        .sort
        drop F{'i'-2};
        skip F{'i'-1};
        Local F'i' = F{'i'-2}+F{'i'-1};
        print;
    #enddo

   F2 = 2;

   F3 = 3;

   F4 = 5;

   F5 = 8;

   F6 = 13;

   F7 = 21;

   F8 = 34;

   F9 = 55;

    .end

   F10 = 89;
```

The above uses the preprocessor to generate the Fibonacci

sequence rather efficiently. We have kept no more expressions in memory than necessary by dropping whatever isn't needed any more. Note also that the contents of the loop were only printed the first time. These contents are even printed for a so called zero trip loop, ie. a loop that isn't executed due to the values of its parameters.

A preprocessor variable can also be defined directly in a 'define' instruction:

```
#define MAXVAL "10"
Symbols
#do i = 1,'MAXVAL'
 a'i'
#enddo
 ;
Local F =
#do i = 1,'MAXVAL'
 + {2*('i'%2)-1} * a'i'
#enddo
 ;
print;
.end

F =
   a1 - a2 + a3 - a4 + a5 - a6 + a7 - a8 + a9 -
   a10;
```

The above example composes the necessary declarations and then sets up a sequence in a number of variables. This is all a function of a single parameter MAXVAL. The 'value' that is assigned to a preprocessor variable is a string, enclosed between double quotes. This way the string may contain nearly any legal characters that can occur in a FORM program. In particular MAXVAL could even be an entire set of statements! In the above

example we used a string that had also a numerical interpretation, enabling us to use it as a cutoff for a do loop. There is another novelty in this little program, although it is well hidden: The object `{2*('i'%2)-1}` becomes either +1 or -1. In the first case there is no problem as it is inserted as the string 1. In the other case it is inserted as the string -1 leading to the string +-1 in the input. FORM will accept strings of plusses and minusses. The combined result will be plus if there is an even number of minusses and minus if there is an odd number of minusses.

The opposite of the 'define' instruction is the 'undefine' instruction. It has only a single argument which should be the name of a preprocessor variable without the single quotes. This variable is then removed from the list of preprocessor variables.

Let us now turn our attention to the do loop. The loops we saw thus far had two parameters after the equals sign. The first is the start value and the second is the last value for the loop parameter. In this case the increment is taken to be one. If the user prefers a different increment he should specify its value as a third parameter:

```
#define MAXVAL "10"
Symbols
#do i = 1,'MAXVAL',3
 a'i'
#enddo
 ;
Local F =
#do i = 1,'MAXVAL',3
 + {2*('i'%2)-1} * a'i'
#enddo
 ;
print;
.end
```

```
F =
   a1 - a4 + a7 - a10;
```

It is also allowed to use negative values for any of the three parameters. The main restriction is that they must be short integers. For most computer systems this means that their value must be in the range -32768 to 32767. The same holds for the evaluation of preprocessor quantities inside the curly brackets.

There is a second type of do loops that has symbolic values for its loop parameter, rather than numerical values. This type of loop is called a listed loop:

```
Symbols e,mu,tau,u,d,s,c,b,t,W,Z,h,gamma;
CFunction V;
Local Vac =
#do i = {e,-1|mu,-1|tau,-1|
  u,2/3|d,-1/3|s,-1/3|c,2/3|
  b,-1/3|t,2/3|W,1|Z,0|h,0|gamma,0}
 + V('i')
#enddo
 ;
print;
.end

Vac =
   V(e, - 1) + V(mu, - 1) + V(tau, - 1) + V(u,2/3)
    + V(d, - 1/3) + V(s, - 1/3) + V(c,2/3) + V(b,
    - 1/3) + V(t,2/3) + V(W,1) + V(Z,0) + V(h,0)
    + V(gamma,0);
```

The parameter field is enclosed by curly brackets and the parameters are separated by the straight line symbol `|`. This way it is possible to use nearly any character inside these parameters

without fear of premature interpretation of the character. If a character like the straight line is to be used it should be preceeded by a backslash character (\).

The next preprocessor feature is the if instruction. It is followed by a single expression without parentheses, and if this expression is true the lines till the matching else or endif instruction are processed. If the condition is false and there is an else instruction then the lines after it till the matching endif are processed:

```
Symbols a0,a1,a2,a3,a4,a5,a6;
Local F =
#do i = 0,6
#if {'i'%3} == 0
 - a'i'
#else
 + a'i'
#endif
#enddo
 ;
print;
.end

F =
    - a0 + a1 + a2 - a3 + a4 + a5 - a6;
```

There can be only one conditional after the `#if` If more of them are needed one should consider using nested if instructions. This restriction may be lifted in future versions. The conditions that can be used may contain:

= or ==
To indicate equality.

!=
: To indicate inequality.

>
: To indicate 'greater than'

<
: To indicate 'less than'

>=
: To indicate 'greater than or equal'

<=
: To indicate 'less than or equal'

So the above program could just as easily have been:

```
Symbols a0,a1,a2,a3,a4,a5,a6;
Local F =
#do i = 0,6
#if {'i'%3} >= 1
 + a'i'
#else
 - a'i'
#endif
#enddo
 ;
print;
.end

F =
    - a0 + a1 + a2 - a3 + a4 + a5 - a6;
```

The final major feature of the preprocessor concerns procedures, or stored sequences of instructions and statements. These

procedures are very much like macro's but in their use they can often look like dynamically linked subroutines. Procedures can be defined either at the beginning of the file in which the program itself is, or in a separate file. This file must then have the name of the procedure and the extention .prc so that FORM can pick it up during run time. Let us first concentrate on the procedures that are defined in the text of the current program. Our differentiation program of section 5 can now be rewritten as a procedure:

```
#procedure diff(x,dx)
 id  g?commuting?noncommuting('x') = g('x');
 multiply,left,'dx';
 repeat;
     id,'dx'*g?noncommuting[n]('x') =
         derivative[n]('x')+g('x')*'dx';
     id,[-sin]('x') = -[sin]('x');
 endrepeat;
 id  'dx'*'x'^n? = n*'x'^(n-1);
 id  f?noncommuting?commuting('x') = f('x');
#endprocedure
*
* The following statements could be part of a
* standard include file:
*
Symbol  x,y,n;
CFunctions sin,cos,exp,g;
Functions [sin],[cos],[-sin],[exp],f,dx;
Set   commuting:sin,cos,exp;
Set   noncommuting:[sin],[cos],[exp];
Set   derivative:[cos],[-sin],[exp];
*
* And now the program:
*
```

```
 Local expr = x^3*(sin(x)+cos(x)*exp(x));
 #call diff{x|dx}
 print;
 .end

expr =
   3*sin(x)*x^2 - sin(x)*exp(x)*x^3 + cos(x)*x^3
    + cos(x)*exp(x)*x^3 + 3*cos(x)*exp(x)*x^2;
```

The procedure starts with the #procedure instruction which has the formal parameters listed as in any 'normal' computer language. These parameters become then preprocessor symbols. The use of these parameters is entirely according to the rules that hold for preprocessor variables. The procedure is terminated with the #endprocedure instruction. The whole procedure is then available for repeated use. Of course we still need the complicated declarations, but as the commentary says they should be part of a standard include file. The final program is now very short. The procedure is invoked with the #call instruction. The parameters obey the same rules as the parameters in the listed loop. If there is a mismatch in the number of parameters FORM will issue an error message and terminate execution immediately.

The eventual version of our differentiation program should have the procedure in a file 'diff.prc' and the declarations in the file 'diff.h'. The declaritions of x,y and dx can be kept outside this file. So 'diff.h' contains:

```
#-
Symbol n;
    CFunctions sin,cos,exp,g;
    Functions [sin],[cos],[-sin],[exp],f;
    Set   commuting:sin,cos,exp;
    Set   noncommuting:[sin],[cos],[exp];
    Set   derivative:[cos],[-sin],[exp];
```

```
#+
```

The program and the output become now:

```
#include diff.h
#-

Symbol y;
Function dy;
Local expr = y^3*(sin(y)+cos(y)*exp(y));
#call diff{y|dy}

print;
.end

expr =
   3*sin(y)*y^2 - sin(y)*exp(y)*y^3 + cos(y)*y^3
    + cos(y)*exp(y)*y^3 + 3*cos(y)*exp(y)*y^2;
```

Note that there is no reason to keep calling our variables 'x' and 'dx' as they are entirely formal parameters. When the user starts constructing this kind of differentiation procedures to his own whishes he should try to avoid using simple names like n,f and g for the internal variables of a procedure. This could easily lead to conflicts with variables that he may choose to use outside his procedures.

Procedures, do-loops and include files may be nested. Together with the nesting of preprocessor variables there can be a fixed (installation dependent) maximum to this nesting, just as there is a maximum to the number of preprocessor variables. If any of these restrictions becomes an obstacle they can be altered by the setting of some parameters in a file named the setup file. Its default name is 'form.set'. More about this file and the parameters that can be set with it is found in the chapter on the setup file.

This chapter is rather technical though. The beginning user will rarely run into these restrictions so he doesn't need to bother.

As we have seen before the listing of the input can be turned off with the `#-` instruction and turned on again with the `#+` instruction. The default is that the input is listed and also the contents of the include files that are read at the 'ground level'. The contents of do-loops are listed only the first time. The contents of procedures are not listed unless a `#+` instruction is inserted. This will never make the #procedure instruction visible because that must be the first line of a procedure. It is possible to make the contents of a loop visible each time the loop is executed by including the `#+` as its first instruction.

2.8 Flow control

In one of the earlier sections we saw the if instruction of the preprocessor. It can determine whether a set of statements is present. These statements will then be executed for all active expressions. Sometimes it is handier to have some statements only executed for a subset of the terms of the active expressions. For that there is the if-statement. This is a fulfledged statement with composite conditions, a possible else and an endif statement. As all statements it has to be terminated by a semicolon:

```
Symbols a,b,i,j;
CFunction g;
Local F = g(i)*b + g(j)*b;
if ( match( g(j) ) > 0 );

 id b = a;

endif;
print;
.end

F =
   g(i)*b + g(j)*a;
```

The above is a simple example of how one term gets selected for the substitution statement. The condition that caused the selection deserves some study. FORM has its attention focused on a single term when it encounters the if-statement. Therefore the condition can only contain questions about the contents of this single term. There are of course a whole many things that one can ask about the contents of a term. In the current version of FORM three basic 'questions' have been implemented. They are:

match

Match is a function of which the argument can be any legal left hand side of an id-statement, including possible options like the select option we saw before. The 'id' is not present and neither is the equals sign. The return value of this function is the number of times that the given pattern can be taken from the term. In the above example the first term has a value of zero for `match(g(j))`, while the second term returns the value one.

count

The count function is for power counting. Its power counting includes vectors and functions. This function has pairs of arguments. For each pair the first argument is an object like a symbol, a function, a vector or a dotproduct. The second argument is a numerical value which is added to the 'count' for each occurrence of a positive power of the object, and subtracted from the power for each occurrence of a negative power. Dotproducts are handled as symbols in the sense that FORM will not look at the individual vectors that make up the dotproduct. Vectors are kind of special as they can occur in so many different ways as we will see shortly. The returned value is the accumulated 'count'.

coefficient

This function has no arguments (even the parentheses are not allowed). It returns the value of the coefficient of the term.

In addition the conditions may contain numbers like the zero in the above example. The count function is mostly used for powercounting purposes as in the next example:

```
V p1,p2,p3,p4,k;
```

```
I mu,nu;
Local F =
 +(p1(mu)+k(mu))*(p2(mu)+k(mu))
 *(p3(nu)+k(nu))*(p4(nu)+k(nu))
 *(1/k.k+1/k.p1-1/k.p2-1/p1.p2);
if ( count(k,1) > 0 );
 discard;
endif;
print;
.end

F =
   p1.p2*p1.k^-1*p3.p4 + p1.p2*p1.k^-1*p3.k +
   p1.p2*p1.k^-1*p4.k - p1.p2*p2.k^-1*p3.p4 -
   p1.p2*p2.k^-1*p3.k - p1.p2*p2.k^-1*p4.k + p1.p2
   *p3.p4*k.k^-1 + p1.p2*p3.k*k.k^-1 + p1.p2*p4.k*
   k.k^-1 + p1.p2 - p1.k*p2.k^-1*p3.p4 + p1.k*
   p3.p4*k.k^-1 + p1.k*p3.k*k.k^-1 + p1.k*p4.k*
   k.k^-1 + p1.k^-1*p2.k*p3.p4 + p2.k*p3.p4*k.k^-1
    + p2.k*p3.k*k.k^-1 + p2.k*p4.k*k.k^-1;
```

If we take the above expression F and then assume that the vector k is so small that any term that has effectively one or more positive powers of k can be ignored we can trow those terms away as was done in the above program. The function count does the power counting for us and if there is a positive count the discard statement is executed. This statement is very simple: It throws the term away!

The above examples could still have been programmed without the use of the if-statement (but with more work and less clarity though). For the next one this isn't possible.

```
S a,b,i,j;
Local F = sum_(i,0,4,a^i/fac_(i))
```

```
    *sum_(j,0,4,b^j/fac_(j));
  if ( coefficient < 1/100 );
   discard;
  endif;
  print;
  .end

F =
   1 + 1/48*a^4*b^2 + 1/24*a^4*b + 1/24*a^4 + 1/36
   *a^3*b^3 + 1/12*a^3*b^2 + 1/6*a^3*b + 1/6*a^3
    + 1/48*a^2*b^4 + 1/12*a^2*b^3 + 1/4*a^2*b^2 +
   1/2*a^2*b + 1/2*a^2 + 1/24*a*b^4 + 1/6*a*b^3 +
   1/2*a*b^2 + a*b + a + 1/24*b^4 + 1/6*b^3 + 1/2*
   b^2 + b;
```

The function `fac_` is the built in factorial function. It is evaluated immediately when its argument is a nonnegative integer. With the if-statement we have placed a cutoff on the size of the coefficients in our expression.

The operators that are recognized in the conditions are the same as in the preprocessor if-instruction:

==
: To indicate equality.

!=
: To indicate inequality.

>
: To indicate 'greater than'

<
: To indicate 'less than'

>=
: To indicate 'greater than or equal'

<=
: To indicate 'less than or equal'

Contrary to the preprocessor if, the if-statement can have composite conditions:

```
Symbols a,b,i,j;
Local F = sum_(i,1,4,a^i)*sum_(j,1,4,b^j);
if (   ( ( match(a) > 2 ) && ( match(b) <= 2 ) )
    || ( ( match(b) > 2 ) && ( match(a) <= 2 ) )
   );

 discard;

endif;
print;
.end

F =
   a^4*b^4 + a^4*b^3 + a^3*b^4 + a^3*b^3 + a^2*b^2
    + a^2*b + a*b^2 + a*b;
```

To allow the composition of these conditions there are some extra operators:

&&
: To indicate a logical 'and'.

||
: To indicate a logical 'or'.

Each element of the composite condition must be enclosed between parentheses, as the built in rules for evaluation are strictly from left to right. There is no hierarchy among the operators!

The most complicated part of the if-statement is the handling of the vectors in the count function. Vectors can occur as loose vectors with an index, as part of a dotproduct, contracted with a function argument that was an index and as a regular (not necessarily linear) function argument. There are of course even more ways but those cannot be considered at all by the count function. It is possible to differentiate between the various occurrences:

```
Vectors k1,k2,k3,k4,k5;
Functions f1,f2,f3,f4;
Set fff:f1,f2,f3;

    count(k1+v,1,k2+d,-1,k3+f,2,k4+vd,-1,k5+?fff,-2)
```

The above count function has the following result:

- The vector k1 has count-value one and is only counted when it occurs as a loose vector with an index.
- The vector k2 has count-value -1 (the countvalue must always be an integer). It will only be counted when k2 occurs inside a dotproduct.
- k3 will only be counted inside one of the two special built in functions that are known to be linear in all its vector arguments, since these arguments are actually indices that were contracted with the index of a loose vector. These two special functions are e_ (the Levi-Civita tensor) and g_ (the Dirac gamma matrix).
- k4 will be counted when it occurs either as a loose vector or inside dotproducts (options can be combined).

- k5 will be counted when it occurs as an argument of one of the functions in set 'fff'. The user indicates with this notation that those occurrences of k5 are due to the contraction of an index. There are no provisions to differentiate between the properties of different arguments.

It is allowed to let one object occur more than once in the argumentlist of the count function. The effects are cumulative.

The if statement can be nested to 10 levels deep. Also the bracketting to construct 'subconditionals' as in the example with the logical 'ands' and 'if' is bound to 10 levels. In practice this rarely causes problems. Inside the body of an if-statement (or its else part) there cannot be end-of-module instructions. The whole range of the if-statement must be part of a single module.

When there is an if-statement there is also use for labels and goto statements. This enables the user to set up any type of loop that he likes, or (more often) to save on much typing work:

```
if ( count(a,1,b,-1) > 0 );
 id a^2 = 1;
 if ( count(c,1,d,-1) < 0 );
 goto 1;
 endif;
else;
 if ( count(a,1,d,-1) > 0 );
label 1;
 id  -----very long set of statements-----
    else;
     id a/d = 1;
    endif
endif;
```

In the above example we could have avoided the use of labels. It served more as a demonstration of the fact that it is allowed

to jump inside the range of an if-statement. When constructing loops it is harder (or even impossible) to avoid labels and goto's:

```
label 1;
----- a number of statements
if ( count(a,1,b,2) > 5 );
goto 1;
endif;
```

The above is a makeshift constructing that imitates a do-while loop. The while loop is made with:

```
label 1;
if ( --condition-- );

---statements---

goto 1;
endif;
```

There is a number of restrictions with respect to labels. A goto statement and its corresponding label must be in the same module. The labels have a number that may range from 0 to 20. The labels are reusable in the sense that a label that has been used in one module can be used again inside the next module. It is not allowed to use the goto statement to jump into the range of a repeat statement.

2.9 Special objects and commands

Thus far we have seen most of the general capacities of FORM. Occasionally we saw a special function like the functions sum_ or e_. Here we'll have a look at what other built-in objects are currently available. We will make one exception though: everything that has to do with gamma matrices is kept for the next section.

In addition to the function sum_ there is the function sump_. This is a summation function which has a slightly different rule for the elements of the sum. The last argument of sum_ is the formula for each term in the sum. In sump_ the last argument is the quotient of the i-th term in the sum divided by the (i-1)-th term. The normalization is such that the first term in sump_ is always one:

```
Symbols x,i;
Local F = sump_(i,0,5,x/i);
Local G = sump_(i,3,5,x/i);
print;
.end

F =
   1 + 1/120*x^5 + 1/24*x^4 + 1/6*x^3 + 1/2*x^2 +
   x;

G =
   1 + 1/20*x^2 + 1/4*x;
```

The properties of sump_ are demonstrated best with the expression 'G'. The first term is always one, so that takes care of i = 3. The i = 4 term is then x/4 times the i = 3 term, and the i = 5 term is x/5 times the i = 4 term. This is very handy for expansions that look like exponential expansions as is shown

with expression 'F'. There should be a word of caution. If the last argument in sump_ contains more than one term and the sum contains a large range of 'i' then the evaluation can become very timeconsuming. There is no intermediate evaluation of the elements in the sum. This means that if the last argument contains two terms and the sum contains 21 elements the evaluation of the last element in the sum involves 1048576 terms!

```
Symbols x,y,i;
Local F = sump_(i,0,10,x/i+y);
write statistics;
.end
```

```
Time =      16.12 sec    Generated terms =       2047
               F         Terms left      =         66
                         Bytes used      =       1258
```

There are two types of delta functions. The first type is the delta with two indices which serves as a metric tensor. Its symbol is d_ and its use is in the field of vector and tensor algebra. When one of its indices is contracted with an index elsewhere in the term the d_ can be removed:

```
Indices mu,nu;
Vector  v;
Local F = v(mu)*d_(mu,nu);
print;
.end
```

```
F =
   v(nu);
```

When the d_ function has two identical indices and they are to be summed over the d_ can be replaced by the dimension of

the space in which these indices are defined. This has been shown already in an earlier section.

The other delta function is indicated by the string 'delta_' and it should have either one or two arguments. If it has one argument the function is replaced by one if the argument is zero and it is replaced by zero if the argument is unequal to zero. If the argument is still symbolic the function is left as is:

```
Symbols x,y,z,n;
Function f;
Local F = f(x) + f(y) + f(z)*x;
id f(z?) = f(z)*delta_(z-y);
id x^n? = delta_(n);
print;
.end

F =
   f(x)*delta_(x - y) + f(y);
```

The delta_ with the argument y-y has been replaced by one. In the second substitution the delta_(1) has been replaced by zero. When the delta_ has two arguments the rules are very similar. If the arguments are equal the delta_ is replaced by one. Are the arguments numeric and unequal then the delta_ is replaced by zero. If the arguments are unequal and at least one of the arguments is still symbolic the delta_ is left untouched:

```
Symbols x,y,z,n;
Function f;
Local F = f(x) + f(y) + f(z)*x;
id f(z?) = f(z)*delta_(y,z);
id x^n? = delta_(n,0);
print;
.end
```

```
F =
   f(x)*delta_(y,x) + f(y);
```

As one can see: the effect is quite the same. FORM doesn't try to symmetrize the delta_ with two arguments. The exchange of the arguments of delta_ would be identical to a change of sign in the delta_ with one argument. There are cases in which this isn't called for.

The next special function is the theta function theta_. Again this function can have either one or two arguments (for a larger number of arguments the special properties are not applied). If it has a single argument and this argument is numeric the theta_ is replaced by zero if this argument is negative and by one otherwise. If the argument is still symbolic the function is left untouched. With two arguments the function is looked at from a symmetrization viewpoint. If symmetrization would require the arguments to be exchanged the function is replaced by zero. If the arguments are already in the proper order the function is replaced by one.

```
Symbols x,y,n;
CFunction g;
Local F = g(x,4);
repeat;
 id g(x,n?) = x^n + g(x,n-1)*theta_(n);
endrepeat;
print;
.end

F =
   1 + x^4 + x^3 + x^2 + x + x^-1;
```

The above shows one of the main uses of the theta_ function:

the cutoff on the definition of recursion relations. Note that the cutoff becomes only active when n is negative!

We have seen the factorial function already. It is represented by fac_ and a single argument. If this argument is a nonnegative integer FORM will replace the function by the corresponding factorial.

The next interesting quantity is the symbol i_. It is the basic imaginary quantity, so its square is -1. Currently complex variables can be declared and FORM knows the difference between a complex symbol and its complex conjugate (e.g. x and x#). Complex conjugation hasn't been built in yet. Therefore the user is advised to use real objects only and use a notation of the type `a+i_*b` when he has to use complex arithmatic.

```
 Symbols x,y;
 Local F = (x+i_*y)^2;
 print;
 .sort

F =
   2*i_*x*y + x^2 - y^2;

 id i_ = -i_;
 print;
 .end

F =
    - 2*i_*x*y + x^2 - y^2;
```

The id statement that is used here serves as a poor mans complex conjugation. The variable i_ can be used for many things where an ordinary symbol can be used. It is illegal to use it as a

wildcard. If it is used in the power of an object this power should be enclosed by parentheses to avoid possible confusion.

There is a set of indices with a numerical value which has been built in. These are called the 'fixed indices'. Normally their value can be any number from zero to 127 inclusive. If the user wants to change this range he should consult the chapter on the setup file. These fixed indices have special properties. They are considered to be nonsummable, i.e. a repeated use of such an index is not seen as a summation over this index. If such an index occurs twice in the same d_ function the d_ function is replaced by one (this is the default value). This value can be changed with the 'FixIndex' statement:

```
Symbol a;
FixIndex 1:-1, 2:6;
Local F = d_(0,0) + d_(1,1)*a + d_(2,2)*a^2;
print;
.end

F =
   1 + 6*a^2 - a;
```

The syntax is rather clear: There are pairs of numbers. The pairs are separated by comma's. Inside a pair the numbers are separated by a colon. The first number of the pair is the number of the index and the second number is the value that FORM should insert when this index occurs twice inside the same d_. When a d_ function occurs with two different fixed indices it is replaced by zero. This implicates that if the user would like to introduce a metric tensor that is offdiagonal he should define his own function for it.

There is a special statement for partial fractioning. It isn't as powerful as it should be yet, but in most cases the user can use

some simple commands to bring his formulae in such a shape that the 'ratio' statement can deal with it. The ratio statement has three symbols as its arguments. The fractions are the first two arguments and the third argument should represent the difference of the second and the first argument. Formally this yields:

```
Symbols a,b,c;
Local F = 1/a/b;
ratio,a,b,c;
print;
.end

F =
   a^-1*c^-1 - b^-1*c^-1;
```

In practice this is rather cumbersome to read and very error-prone. The notation that allows the use of square braces to enclose a name was invented to avoid this notational problem:

```
Symbols [x+a],[x+b],[b-a],x,a;
Local F = 1/[x+a]/[x+b];
Local G = x^3/[x+a]^2;
ratio,[x+a],[x+b],[b-a];
ratio,x,[x+a],a;
print;
.end

F =
   [x+a]^-1*[b-a]^-1 - [x+b]^-1*[b-a]^-1;

G =
   3*[x+a]^-1*a^2 - [x+a]^-2*a^3 + x - 2*a;
```

This time it is rather easy to read what is happening, even though we are still manipulating only symbols. The ratio com-

mand becomes active when at least one of the first two symbols in its parameter field occurs with a negative power in a term and the other is also present with some power. It should be noted that the answer of a partial fractioning operation isn't always unique. This can be seen in the answer for G. Rather than `x-2*a` it could also have contained `[x+a]-3*a`. Such ambiguity will only occur with positive powers, so it is rather easy to correct for it with straight forward substitutions.

The last statements we will discuss here concern argument fields of functions. Currently only the properties of the built in functions are known to FORM. This means that FORM can take the antisymmetry of the Levi-Civita tensor into account. The user can force the symmetrization or the antisymmetrization of the argument field of a function with the 'symmetrize' or the 'antisymmetrize' statement. The simplest form of these commands is with a single argument: the function to be acted upon:

```
Symbols x,y;
CFunction g;
Local F = g(x,y)+g(y,x);
symmetrize g;
print;
.end

F =
   2*g(x,y);
```

These commands have several options that can be found in the chapter on functions.

2.10 Numbers and statistics

Till now we have tacitly assumed that all coefficients and powers that we used would cause no problems whatsoever. In practice computers are finite so there will always be limitations. In addition there are the 'voluntary' limitations belonging to the language of FORM. The main limitation of this type is that FORM does its arithmatic either over the rational numbers or over a finite field in which the arithmatic is modulus a positive integer number. The absence of floating point numbers is experienced by some as a shortcoming, but FORM is neither a program for numerical evaluation, nor a 'mathematicians workbench'. It is a program for the manipulation of formulae, and just as the user would, during the manipulation stages of his computations, give the transcendental numbers a name like pi, he can do so in FORM. For the numerical evaluation stages he can use any of the languages that are designed for computations like fortran or pascal.

Under ordinary circumstances the rational arithmatic is purely for coefficients. Also the arguments of functions can be rational numbers. The limitation on the size of the numerator and the denominator is $2^{1}600$ in most implementations of FORM. For most applications this will be more than enough. In future versions it will be possible to alter this maximum size, using the setup file.

The arithmatic rules for the coefficients can be switched to modular arithmatic with the 'modulus' statement:

```
modulus 7;
Symbol a;
Local F = 20*a + 21*a^2 + 22*a^3 + 1/5*a^4;
print;
.end

F =
   3*a^4 + a^3 + 6*a;
```

The above modulus command forces all arithmatic to be taken modulus 7. This means that also fractions will be converted to integers. So is 1/5 the integer that, if multiplied by 5, will give 1. This happens to be 3 in the above example. Note also that all numbers are converted to the positive range.

In addition to the regular arithmatic the modulus command can also control the limitation of powers by means of the Fermat formula that x^p = x when arithmatic is modulus p:

```
modulus 7;
Symbol a;
Local F = 20*a^-5 + a^12;
print;
.end

F =
   a^6 + 6*a;
```

If the user doesn't like the powers to be reduced like this he should specify a negative number in the modulus command:

```
modulus,-7;
Symbol a;
Local F = 20*a^-5 + a^12;
print;
.end

F =
   a^12 + 6*a^-5;
```

Note that the comma between 'modulus' and '-7' is necessary as a blank space is now not converted to a comma automatically (it shouldn't before the - sign!). Now we know what to do with the positive and the negative values in the modulus statement there

is only one value left unspecified: zero. The statement 'modulus 0' sets arithmatic back to the rational numbers. The modulus statement has the same status as the other declarations. A .global instruction makes it into a global declaration.

The modulus command has one more option. If the number in the modulus command is followed by a colon and a second number all coefficients will be printed as a power of this second number. For this to be possible this second number has to be a 'generator' which means that its powers must generate all possible numbers. These powers have to be stored in a table, so the amount of available memory may put a limitation to the size of the modulus:

```
Symbol x;
modulus 7:3;
Local F = x + 2*x^2 + 3*x^3
    + 4*x^4 + 5*x^5 + 6*x^6;
print;
.end

F =
   3^3*x^6 + 3^5*x^5 + 3^4*x^4 + 3^1*x^3 + 3^2*x^2
    + x;
```

There exist more manipulations with the powers of symbols. When a symbol is declared it is possible to specify a range of powers for it. In that case only powers inside this range will be admitted. All other powers will be considered to yield zero:

```
Symbols x(-4:4),y(:5),z(-5:);
Local F = (1/x+x)^6 + (1+y)^6*y^2
    + z*(1-1/z)^10;
print;
```

```
.end

F =
   10 + 6*x^4 + 15*x^2 + 15*x^-2 + 6*x^-4 + 20*y^5
    + 15*y^4 + 6*y^3 + y^2 + z + 45*z^-1 - 120*
   z^-2 + 210*z^-3 - 252*z^-4 + 210*z^-5;
```

A range is specified between parentheses after the name of the symbol. The minimum power and the maximum power are separated by a colon. If either is absent then the default minimum or maximum is substituted. This is at least +/-10000, but the exact value depends on the implementation. A power which is greater than the default value (or less than minus the default value) results in an error message because this default value is also used when no range of powers is specified.

Many numbers in FORM are restricted to be so called 'short integers'. The exact range of values that these integers can take is implementation dependent. It is at least -32768 to +32767. An example of such integers is the value that one may specify in the fixindex statement. The dimension statement takes positive 'short integers'. The preprocessor arithmetic is also restricted to short integers. This arithmetic is independent of the settings of the modulus command.

Until now we have seen small programs only. Whenever we printed the statistics of a run there was always only one message in the statistics for each expression. When the programs become bigger the run time statistics may become a little bit more confusing too. In the following program some of the parameters of FORM were set in the file form.set so as to illustrate all possibilities in one 'little' program:

```
Vectors p1,p2,p3,p4,p5,p6,p7,p8;
```

```
    Local F = e_(p1,p2,p3,p4,p5,p6,p7,p8)
            * e_(p1,p2,p3,p4,p5,p6,p7,p8);
    contract;
    .end

Time =      20.33 sec    Generated terms =       5289
                F        Terms left      =       2651
                         Bytes used      =     148052

Time =      42.00 sec    Generated terms =      10530
                F        Terms left      =       7028
                         Bytes used      =     398314

Time =      64.99 sec    Generated terms =      15788
                F        Terms left      =      11281
                         Bytes used      =     641010

Time =      88.11 sec    Generated terms =      21061
                F        Terms left      =      15538
                         Bytes used      =     883316

Time =     111.22 sec    Generated terms =      26320
                F        Terms left      =      19833
                         Bytes used      =    1128394

Time =     125.30 sec
                F        Terms active    =      17712
                         Bytes used      =    1006330

Time =     148.91 sec    Generated terms =      31579
                F        Terms left      =      22060
                         Bytes used      =    1254330
```

```
Time =     172.29 sec    Generated terms =      36841
               F         Terms left      =      26441
                         Bytes used      =    1503974

Time =     188.25 sec    Generated terms =      40320
               F         Terms left      =      29382
                         Bytes used      =    1671784

Time =     205.86 sec
               F         Terms active    =      25739
                         Bytes used      =    1629768

Time =     245.73 sec    Generated terms =      40320
               F         Terms in output =      18155
                         Bytes used      =    1024118
```

In this program we compute a large 8 by 8 Gram determinant. Actually in this program we compute only the number of terms in the output, because we throw the output away after the program is finished. Because we have an 8 by 8 determinant we have to generate 8! terms. After 5289 terms however FORM prints spontaneously some statistics. It does so every time one of its buffers is filled. It sorts this buffer and writes the results to another buffer. This is what happens the first four times. The fifth time its also sorts the first buffer, but now the second buffer is full, so the results of the sort cannot be written. FORM sorts now the second buffer and writes the results to file. On home computers one may now note some disk activity. As this operation doesn't involve the generation of new terms this part has not been mentioned in the new statistics that describe how much is left after the sorted results of the second buffer have been written. Now there is space again in the second buffer, so FORM continues. It finishes generation. It then sorts the contents of the second buffer and writes

this also to file. The final sort is the merger of two or more parts in the sort file (the third stage sort). There is now a very intensive disk activity and the speed of the disk has a measurable effect on the running time here. Finally the last statistics show the overall results.

It would of course have been easy to suppress all these statistics and print only the final statistics. Many users prefer the regular printout of intermediate statistics. This allows them to monitor what is happening and to see whether things take an undesired direction.

There are currently only three stages in the sort. In the third stage the number 'stage 2 results' that can be merged is limited to a number that is implementation dependent. On small computers this is at least 32, while on larger computers it should be at least 64. Coupled to the size of the buffers it is usually sufficient to swamp the available disks, so the run would be terminated earlier anyway. In a future version even this limitation will be lifted with a fourth stage sort. The advanced user can play around with the parameters that influence this sorting. He should however take very good notice of the correlations that exist between the various parameters as explained in the chapter on the setup file.

2.11 Dirac matrices

This section of the tutorial contains information that is particular to some fields of physics and mathematics. The person who isn't interested in Dirac gamma matrices should skip this section with having ill feelings that he may have missed something.

FORM has the Dirac gamma matrices as built in functions with the name g_. It knows some properties of these matrices and it has some commands to take the trace of a string of these matrices. In addition there are some extra names gi_, g5_, g6_ and g7_ to make some operations more compact.

```
 Indices m1,m2,m3,m4;
 Local F = g_(1,m1)*g_(1,m2)*g_(1,m3)*g_(1,m4)
  + g_(1,m1)*g_(1,m2)*g_(1,m3);
 print;
 .sort

F =
   g_(1,m1,m2,m3,m4) + g_(1,m1,m2,m3);

 trace4,1;
 print;
 .end

F =
   4*d_(m1,m2)*d_(m3,m4) - 4*d_(m1,m3)*d_(m2,m4)
    + 4*d_(m1,m4)*d_(m2,m3);
```

We see the gamma matrices in expression F with a first argument that must be an index an is in the above case a 1. This index is there to indicate a 'spin line'. Matrices of a different spin line commute with each other and have nothing to do with each other when a trace is taken. When the output is printed we see

that all matrices were raked together into a single g_ function. This is done when no other noncommuting functions are between the matrices. Otherwise there may be more of those strings of matrices. After the printing the 'trace4' command was issued for spin line 1. The trace4 command evaluates the trace of all gamma matrices with spin line 1 in each term. They are taken into one string regardless of whether there are noncommuting functions in between. The user must take care that any possible conflicts are resolved before he issues a trace command. The trace4 statement forces the trace to be taken in 4 dimensions. It uses tricks that are particular to 4 dimensions, allowing a rather fast evaluation of the trace. In the above example it means that the string with three gamma matrices is removed because its trace is trivially zero. The trace of the four gamma matrices gives its canonical form. There is a second trace statement: 'tracen' which takes the trace in an unspecified dimension (which looks rather much like 4 though). In the above example this would not have made much difference. It does in the following example:

```
Symbol n;
Index m1;
Vectors p1.p2;
Local F = g_(1,m1)*g_(1,p1)*g_(1,m1)*g_(1,p2);
trace4,1;
print;
.sort

F =
    - 8*p1.p2;

drop F;
Index m4=n;
Local G = g_(1,m4)*g_(1,p1)*g_(1,m4)*g_(1,p2);
```

```
 tracen,1;
 print;
 .end

G =
   8*p1.p2 - 4*p1.p2*n;
```

Note that replacing n by four makes G equal to F. In general the 4 dimensional trace algorithms are more compact. When there are contracted indices it is possible to apply the Chisholm identities. It is also possible to use reduction formulae in the Dirac algebra because the exact structure of the algebra is known. In n dimensions things are necessarily somewhat vague, so many of these handy algorithms are not allowed.

The use of the axial matrix γ_5 is allowed. Its symbol is g5_ and it has only the spin line index. If it is inserted directly in a string of gamma matrices it can be put in as 5_. In addition there are the matrices $\gamma_6 = 1 + \gamma_5$ and $\gamma_7 = 1 - \gamma_5$. These can be used as g6_ and g7_ or a 6_ and a 7_ inside a string of gamma matrices. The unit matrix is gi_ with just a spin line index.

Example: Muon decay.

```
 Vectors pmu,pmuneutrino,pe,peneutrino;
 Indices m1,m2;
 Symbols emass,mumass;
 write statistics;
 Local M =
   g_(1,pmuneutrino)*g_(1,m1)*g7_(1)
  *(g_(1,pmu)+mumass)*g_(1,m2)*g7_(1)
  *(g_(2,pe)+emass)*g_(2,m1,7_)
  *g_(2,peneutrino)*g_(2,m2,7_);
 trace4,1;
 trace4,2;
```

```
    contract 0;
    print;
    .end

Time =       0.19 sec    Generated terms =         13
                M        Terms left      =          1
                         Bytes used      =         26

   M =
      256*pmu.peneutrino*pmuneutrino.pe;
```

The above program computes the matrix element for the decay of a muon into an electron, a muon neutrino and an anti electron neutrino. There are two spin lines and we have used the Fermi four point interaction. In the second spin line we have put the γ_7 together with the other gamma matrix of the vertex in one string of gamma matrices. Because there are two strings we have to issue two trace commands. After this there may be terms with Two Levi-Civita tensors, so we contract those. The final answer is the familiar result that can be found in many textbooks.

In some cases one would like to have gamma matrices that don't have a spin line index but rather the two indices that indicate that it is a matrix in spinor space. It is rather easy for the user to define such a function himself. To convert a string of such matrices into a string of gamma matrices so that its trace may be taken isn't very difficult:

```
    Vectors p1,p2,p3,p4;
    Indices m1,m2,m3,m4;
    CFunction g;
    Local F = g(m1,m2,p1)*g(m4,m1,p4)
             *g(m2,m3,p2)*g(m3,m4,p3);
    repeat;
```

```
 id g(m1,m2?,??)*g(m2?,m3?,???)
 = g(m1,m3,..,...);
endrepeat;
print;
.sort

F =
   g(m1,m1,p1,p2,p3,p4);

repeat;
 id g(m1,m1,m2?,??) = g_(1,m2)*g(m1,m1,..);
endrepeat;
id g(m1,m1) = gi_(1);
print;
.sort

F =
   g_(1,p1,p2,p3,p4);

trace4,1;
print;
.end

F =
   4*p1.p2*p3.p4 - 4*p1.p3*p2.p4 + 4*p1.p4*p2.p3;
```

The above program shows all techniques. First the function g is used as the gamma matrices with two indices. The matrices are raked together in one string that is clearly the trace of a product of matrices. Now the easiest would be to use a command like:

```
id g(m1,m1,??) = g_(1,..);
```

but alas this form of wildcarding cannot be used for the functions d_, e_ and g_. Therefore we make a loop in which we pull

out one gamma matrix at a time. It is essential that this is done from the left. This way we get no problems with the fact that the gamma's are noncommuting functions. The noncommuting functions are always to the left of the commuting functions when FORM normal orders its terms. This produces indeed the desired string of gamma matrices. After this it is easy to take the trace. One should also note that we didn't wildcard the index m1. This enables us to select the matrices of a single spin line only. It avoids that we mess up the matrices of two spin lines.

For more information about the gamma matrices and what algorithms are used for taking the traces one should consult the chapter on gamma matrices.

Running FORM

On most systems FORM is invoked by typing the name of the program file (usually form, but it could have a path in front of it). After the name there may be some parameters. The set of parameters is called the 'command tail'. The last parameter is usually the name of the file that is to be executed by FORM. Before this file name there may be some options. These options can be given by their full name or a part of the name. Usually (except for llog) the first character is already sufficient. The options are case insensitive. They are:

-check

This flag tells FORM to check the syntax of the program only. It is mainly intended for long programs. By running them first with this flag the user can avoid the unpleasant situation of having the program abort on a syntax error near the end of the program.

-log

This flag should preceed the name of the file that contains the input for FORM. Normally the output is either written to the screen (not always a good idea if one likes to keep a copy of the output) or to a file (via output redirection). This last mode has the limitation that the user cannot see how well the program is advancing. The -log option gives the best of both by having FORM write the output both to the screen and to a file of which the name is formed by appending the extension .log to the name of the input file. On some systems only one file extension is allowed. On those systems FORM will strip the old extension first before

appending the .log extension.

-llog
Same as -log, but only the last block of statistics is written for each expression. This doesn't affect the display of the statistics on the screen.

-setupfile name
'name' is the name of a file with setup parameters. When FORM begins execution it looks first whether there is a file 'form.set' in the current directory. If this is the case the -S parameter is ignored, together with its file name. If there is no file 'form.set' and if there is a -setup parameter FORM will try to open the given file and take setup parameters from it. The order in which parameters should be used enables the user to define an alias for FORM that has a standard setup file in it while still being allowed to use a local setup file.

-tempdir pathname
The pathname should point to a directory in which FORM should make its temporary files. On many systems it is advisable to use a 'tempdir'. If the temporary files are made in one of the users directories their size may be restricted by the file quota of the user. There is another way of passing the name of a directory for temporary files to FORM. It is described in the chapter on the setup file.

In addition there is the option '-interactive'. This option activates the interactive mode. In this mode FORM will take its input from the keyboard rather than from an inputfile. This part of FORM is the most sensitive to the type of computer it is running on. Therefore the interactive mode has not yet been completed properly, so it will only run with a handicap and the severeness

of this handicap depends on the machine FORM happens to be running on.

Example:

```
form -l foo
```

This causes FORM to execute the program in file 'foo' The output is written both to the screen and to the file 'foo.log'.

On VMS systems the '-' that starts an option can be replaced by the character '/'. In addition all systems allow the use of '-s=filename' and '-t=pathname' rather than '-s filename' and '-t pathname'.

Command structure

FORM commands are grouped in **modules**. Each module is read in and compiled. If there has been no error the module is executed. A program can consist of as many modules as the user likes. Modules are terminated with a line in which the first relevant character is a period. The period is followed by a keyword which indicates the type of the module. The **case** of the characters of any keyword in FORM is unimportant. The case is only important in the names of variables and maybe in file names. This last sensitivity is a function of whether the computer system on which FORM runs is sensitive to the case of the characters. Most mainframes are sensitive to it, while for instance MSDOS and derivates are not.

Modules consist of **statements** or instructions. The number of statements that is allowed in a module depends on the installation. It is influenced by several parameters that the user can modify if these limitations restrict him. How this can be done is explained in the chapter on the setup parameters. A typical restriction is 100 statements with at most 32767 characters per statement and no more than 50000 bytes in the compiled output of the module.

A statement can be either a declaration, the setting of a flag or an executable statement. All statements start with zero or more blanks (or tabs) and a keyword. Statements must end with a semicolon and can extend over more than one line. Lines that have the character $*$ in column 1 are considered to be commentary. Also all characters after a semicolon in a line are seen as commentary. Many statements start with a keyword that is so general that a part of that keyword is already recognized by FORM. Examples

are L for Local, I for Index etc. The list of mandatory and optional parts of the keywords is given in the chapter 'List of commands'.

Blank space is relevant in FORM. Parameters in a declarative statement may be separated either by white space (blanks, tabs) or by comma's. White space that is adjacent to an operator or a bracket is ignored. So

```
L     F ( a b ) = a * b;
```

is read as:

```
L,F(a,b)=a*b;
```

When arithmatic symbols are involved white space is irrelevant. This means that

```
L     F ( a - b ) = a * b;
```

is read as:

```
L,F(a-b)=a*b;
```

Usually this is what the user wants. This is not the case in the following:

```
modulus -5;
```

which is read as:

```
modulus-5;
```

and an error message will be given. The use of a comma is necessary here:

```
modulus,-5;
```

The built-in rules to separate relevant blanks from irrelevant blanks are almost always in agreement with the intuitive expectation. The one exception is encountered in the representation of very long numbers or names. Because

```
L      F = 12345678
       901234567890;
```

is read as:

```
L,F=12345678,901234567890;
```

One may 'escape' the end of line in the standard UNIX fashion:

```
L      F = 12345678\
901234567890;
```

This will give the proper interpretation:

```
L,F=12345678901234567890;
```

Actually there is no need to start in column one in the second line. After the backslash all spaces and tabs at the beginning of the 'continuation' line are ignored, so

```
L      F = 12345678\
           901234567890;
```

would produce the same result.

Powers are indicated by the character $\wedge$, but the string $**$ is also accepted. Only symbols, dotproducts and subexpressions may have powers which are implemented in a natural way. When other objects have powers there can either be an ambiguity or the object is placed inside the 'exponent function'. This is a function without a name and with two arguments. The second argument

is the exponent. In simple cases FORM can expand the exponent. For the rest its presence is tolerated and it can be printed. Functions with powers are printed as a sequence of individual occurrences of the function. The same holds for vector components.

Apart from the 'compiler' commands there are the preprocessor commands. The preprocessor may operate on the code before it is offered to the compiler. It is the preprocessor that tries to interpret the blanks and replaces the string $**$ by $\wedge$. In addition there are several preprocessor instructions, each of which starts with the symbol #, followed by a keyword. Such an instruction may cause the definition of preprocessor variables, the duplication of code in a range while substituting preprocessor variables in this code, inserting the contents of a file and more. Preprocessor statements are not terminated by a semicolon. They occupy either a single line (which can be extended with the use of a backslash at the end of the line), or a range which is enclosed by curly braces (some statements only).

Preprocessor variables can be recognized easily as they must be enclosed between single quotes when they are used. This allows the user to concatenate the contents of preprocessor variables to form longer strings. The exact preprocessor syntax can be looked up in the chapter on the preprocessor. Mastering the preprocessors possibilities is important when one would like to take FORM to the limits of its capabilities.

There are several conventions regarding the input and the output. A vectorproduct is written as p1.p2 when p1 and p2 are the two vectors. The alternative is to write p1$p2 as FORM will use this notation when writing output in Fortran format.

For contracted indices we use the Schoonschip notation. When a vector is written at the position where an index is expected this means that the index that would have been at that location was

the same as the index of the vector, and that the index has been summed over according to the Einstein summation convention. This notation is also very useful in hand computations.

Expressions

The main difference between numerical computations and formula manipulations lies in the order of the action. In numerical computation one first computes all numbers that are needed to evaluate the statement that leads to the final result. In computer algebra things work exactly the opposite: A formula is defined, after which operations on this formula may lead to a different formula. One has to first define the relation that would have been used last in the numerical computation, then introduce the next to last relation and finally end with the relation that would be evaluated first when working numerically. In other words: in numerical computation one starts with the components while in formula manipulations one ends up with an expression that consists of components.

```
*     Define the expression F:
L     F = a + b;
*     Give the values of a and b:
id    a = 2;
id    b = 5;
```

This would be in fortran:

```
      a = 2;
      b = 5;
      F = a + b;
```

A formula is called an expression. Expressions are always defined first, afterwhich the operations that work on these expressions are introduced. So a module may consist of declarations, definitions of expressions and operations on these expressions.

Anywhere in the module there may be commentary and the settings of run time parameters (like print flags, bracket information etc). One should however realize that there are two types of expressions: **Active expressions** and **Stored expressions**. The active expressions are the object of the operations. This means that each operation is executed on each expression that is active in the given module. A stored expression is an expression that cannot be the object of operations any more, but its contents can be used. The active expressions are again divided into two classes: **Local expressions** and **Global expressions**. Local expressions are expressions that stay active till the next .store or .end termination of a module. After the .store or .end instruction has been executed these expressions are removed from the system. The global expressions stay also active till the next .store or .end statement, but after a .store instruction they are copied to a special file, so that they may be used as stored expressions. They will remain stored expressions for the rest of the program or until a 'Delete Storage' statement is encountered in which case all stored expressions are deleted.

```
s     a,b,c;
L     F = (a+b)^2;
id    b = c - a;
write names;
print;
.sort
id    a = b + c;
print;
.store
s     b,c,d;
L     F = (c-d)^2;
write names;
print;
```

```
.end
```

In this example F is declared to be a local expression that is equal to `(a+b)^2`. The substitution `b -> c-a` is made and the results are sorted and printed. In the next module the original F is still an active expression and the substitution `a -> b+c` is made. After this the expression is printed and forgotten (because of the .store instruction). It is removed from all lists, so one is allowed to reuse the name F for a different expression after the .store as is done in this example.

```
s     a,b,c;
g     F = (a+b)^2;
write names;
.store
L     G = F*F;
write names;
print;
.end
```

The expression F is declared to be global, evaluated and stored after which itis removed from the list of active expressions. The local expression G is defined and its definition uses the stored expression F. Finally G is evaluated and printed, after which the program terminates. The above results could also have been obtained in a simpler program:

```
s     a,b,c;
L     F = (a+b)^2;
write names;
.sort
drop F;
L     G = F*F;
write names;
```

```
print;
.end
```

In this case the expression F doesn't have to be written away as it stays a local expression. The drop statement inactivates F for the current module and will remove it from all lists at the end of the current module.

It is allowed to have more than one active expression. Only the sum of all expressions that are currently in the namelists is subject to restrictions. These limitations have a default value (typically 100 or 200) but they can be altered with the variable 'Expressions' in the setup file.

```
s     a,b,c;
.global
L     F1 = (a+b)^2;
L     F2 = (a-b)^3;
id    b = 2*b+c;
.sort
skip F1,F2;
L     F3 = (a+c)^5;
.sort
skip F1,F2,F3;
L     F4 = (a+b)^2;
L     F5 = (a-b)^3;
id    b = b+2*c;
.sort
drop F1,F2,F3;
skip F4,F5;
L     F6 = F1*F2-F3;
.sort
drop F4,F5,F6;
L     h = F6*F4*F5;
print
```

```
.end
```

The above example shows a useful technique: The parts of an expression are evaluated separately and then the entire expression is put together. This can often save much computer time. When this technique is used the expressions are treated again like the numbers in a Fortran program. The skip statement inactivates the expressions that are mentioned in it for the current module. This means that they can still be used in the right hand side of a statement, but they will not be operated upon.

Often one may need only a part of an expression. A popular example is formed by an expression that is quadratic in a given variable (say x) and one needs to know the discriminant of the expression. This can be done by bracketing with respect to the interesting variable (see the chapter on brackets) and then referring to the contents of the brackets using the notation of the straight brackets (`[` and `]`). Whenever one refers to `f[x^2]` FORM will substitute that part of f that has exactly `x^2` outside parentheses. So if `x^2*y` would be outside parentheses the contents of that bracket would not be part of the substitution.

```
s     a,b,c,x,y,z,s,t,u;
.global
L     F = a^2*b + b^2*c + c^2*a - 4*a*b*c;
id    a = x + s - t;
id    b = y + t - u;
id    c = z + u - s;
b     x;
print;
.sort
drop F;
L     G = F[x]^2 - 4*F[x^2]*F[1];
b     z;
print;
```

```
.end
```

This program finds the discriminant of F which is a quadratic form in x. As long as there are no built in instructions for solving sets of equations this use of the contents of brackets can be very profitable when solving equations. For example, if only x was taken outside brackets when E1 and E2 were stored then:

```
E1*E2[x] - E2*E1[x]
```

gives a new equation that has no single x outside parentheses.

Expressions can have parameters. In the expression

```
S    a,b,c,d;
G    F(a,b) = (a+b+c)^2;
```

the symbols a and b are treated as parameters, while the symbol c is a fixed symbol. This is used as follows:

```
S    a,b,c,d;
G    F(a,b) = (a-b+c)^2;
print;
.store
L    G = F(d,b);
print;
.end
```

In this program G becomes the formula (d-b+c)∧2. There are several restrictions when using these parameters. Parameters must be either symbols, indices, vectors or functions. Currently numbers and subexpressions are not allowed. This means that

```
L    G = F((c+d),b);
```

will give an error message. Instead one should use:

```
S    e;
L    G = F(e,b);
id   e = c+d;
```

There must also be an exact match between the number and the types of the parameters. An expression with a different number of arguments or different types of arguments is considered to be an entirely different expression.

Only **stored** expressions can be referred to with parameters. When an expression has been defined with parameters they only become active during the storing process. FORM will read the parameters, independent of whether the expression has been stored, but at the moment of substitution it will ignore them if the expression is still active.

Sometimes one would like to execute a number of operations or substitutions on a subset of the currently active expressions. For this purpose one may use the **skip** statement. It causes the expressions that are mentioned in it to skip all activity in the current module. They can however still be used in the right hand side of another expression or a substitution. In addition there is the **drop** statement. It causes the expressions that are given in it to be skipped during the current module (they can still be used in right hand sides) and when the module is completed the expression is removed completely. The targets of the skip and drop commands must be active expressions. Reference to a stored expression will be ignored. Example:

```
S    a,b,c,d;
L    F1 = (a+b+c)^2;
L    F2 = (a+b+c)^3;
L    F3 = (a+b+c)^4;
b    a;
.sort
```

```
skip F1,F3;
drop F2;
L    F4 = F1*F3[a^2];
L    F5 = F2*F3[a^3];
id   c = d;
print;
.sort
print;
.end
```

In this example all expressions are local. First F1, F2, F3 are defined and the powers of 'a' are taken outside parentheses. In the next module these three expressions will not be acted upon. F4 is the product of F1 and the contents of the a∧2 bracket of F3. F5 is something similar. The replacement c = d is only made in F4 and F5 and they are also the only expressions that are printed. After the .sort F2 is removed from the system, so the final print statement causes the printout of F1, F3, F4 and F5.

Finally there is a feature for interactive use (currently FORM doesn't operate very well in the interactive mode). The commands 'local' or 'global' with just the names of expressions following will alter the local or global properties of these expressions if they were defined properly before.

Variables

The objects of the symbolic manipulations are expressions. Expressions are built up from variables. FORM knows several types of variables, each of which has special rules assigned to it. The types of variables are symbols, vectors, indices, functions, sets and expressions. The expressions are used either in the definition of an expression or in the right hand side of an expression or a substitution. When an expression is used in the right hand side of another expression or a substitution it will be replaced by its contents at the first opportunity. Therefore expressions will never occur as a variable in the output of another expression and we will further ignore their potential presence.

The right hand side of an expression can consist of symbols, vectors, indices and functions. All these objects have to be declared before they can be used. The rules connected to each of these types of variables are described in the sections below. Sets are collections of variables of the same type. They can be used during wildcarding to indicate which object may match a wildcard.

6.1 Names

There are two types of names. Regular names consist of alphabetic and numeric characters with the condition that the first character must be alphabetic. FORM is case sensitive with respect to names. In addition there are **formal names**. These names start with the character `[` and end with the character `]`. Inbetween there can be any characters that are not intercepted by the preprocessor. This allows for the use of variables like `[x+a]`. Using

formal names can improve the readability of programs very much, while at the same time giving the user the benefits of the greater speed. The use of denominators that are composite (like `1/(x+a)`) is usually rather costly in time. Often `1/[x+a]` is equally readable while leading to the same results. Note however that the variable `[x+a]` will have to be declared properly.

Some names may contain special characters. All built in objects have as their last character an underscore (_). Dotproducts (the scalar product of two vectors) consist of two vectors separated either by a period or a dollar sign. The dollar sign is used by FORM when the output of the program has to be fortran compatible. The user may apply either notation. These conventions avoid the possibility of conflicts with reserved names, allowing the user full freedom in his choice of names.

The complex conjugate of a complex quantity is indicated by the character `#` appended to the name of the variable.

The length of names is not restricted in FORM. There is one exception to this rule: names of expressions cannot be longer than 16 characters. There is also a physical limitation with respect to the length of names. The length of the 'NameBuffer' puts a limit on the sum of all characters in all names. The use of this buffer is explained in the chapter on the setup file.

6.2 Symbols

Symbols are plain objects that behave most like normal variables in hand manipulations. Many hand manipulations concern polynomial formulae of simple algebraic variables. FORM assumes that symbols commute with all other objects and have a power connected to them. This power is limited to an installation dependent maximum and minimum. A power outside this range will lead to an error message. The user may override this built in re-

striction by one of his own that is more restrictive. Any power that falls outside the user defined range leads to the removal of the term that contains the variable with this power. Such a power restriction can be defined for each symbol separately.

Symbols can also have complex conjugation properties. A symbol can be declared to be real, imaginary or complex. This property is only relevant when the complex conjugation operator is used. This operator has not yet been implemented.

The syntax of the statement that defines symbols is given by:

```
S[ymbols]    name[#{R|I|C}][(min:max)];
```

Each variable is declared by the presence of its name in a symbol statement. If the # symbol is appended it should be followed by either the character C, I or R to indicate whether the variable is complex, imaginary or real. The #R is not really necessary, as the type real is default. It is not relevant whether the C, I, R are in upper or in lower case. A power restriction is indicated with a range between regular parentheses. If one of the two numbers is not present the default value is taken. This default value is installation dependent, but at least -10000 resp. 10000. Each symbol statement can define more than one variable. In that case the variables have to be separated by either comma's or blanks. Example:

```
S    x,y,z,a#c,b#c,c#c,r(-5:5),s(:20),t#i(6:9);
```

In this statement x, y and z are normal real algebraic variables. The variables a, b and c are complex. This means that for each of these variables two entries are reserved in the property lists: one for the variable and one for its complex conjugate. The variable r has a power restriction: Any power outside the specified range will cause the term containing this power to be eliminated. This is particularly useful in power series expansions. The restrictions

on s are such that there is no limitation on the minimum power of s –with the exception of the built in restrictions– but a term with a power of s that is larger than 20 is eliminated. The variable t is imaginary. This means that under complex congugation it changes sign. Its power restrictions are somewhat uncommon. Any power outside the range 6 to 9 is eliminated. There is however one exception: a term that doesn't contain t to any power (t^0) is not affected.

```
s     x(:10),y;
L     F=y^7;
id    y=x+x^2;
print;
.end

Time =    0.51 sec     Generated terms =          4
             F         Terms left      =          4
                       Bytes used      =         66
   F =
      35*x^10 + 21*x^9 + 7*x^8 + x^7
```

Note that all terms with a power greater than 10 don't even count as generated terms, as they are intercepted immediately after the replacement, before any possible additional statements can be carried out.

6.3 Vectors

A vector is an object with a single index. This index represents a number that indicates which component of the vector is meant. Vectors have a dimension connected to them which is the dimension of the vector space in which they are defined. In FORM this dimension is by default set to 4. If the user likes to change this

default this can be done with the 'Dimension' statement. The use of this command affects the dimension of all vectors and the default dimension of indices. Its syntax is:

```
dimension number;
```

or

```
dimension symbol;
```

The number must be a number that fits inside a FORM word which is an installation dependent size, but it will be at least 32767. The number must be positive. Negative values are illegal. If a symbol is specified it must have been declared before. Any symbol may be used with the exception of i_.

The declaration of vectors is rather straightforward:

```
V[ector] name [,MoreNames];
```

The names of the vectors may be separated either by comma's or by blanks. Example:

```
V    p,q;
I    mu,nu;
L    F=p(mu)*q(nu);
```

6.4 Indices

Indices are objects that represent a number that is used as an integer argument for counting purposes. They are used mostly as the arguments of vectors or multidimensional arrays (or tensors). Their main property is that they have a dimension. This dimension indicates what values the index can take. A four-dimensional

index can usually take the values 1 to 4. A very important property of an index is found in the convention that it is assumed that an index that is used twice in the same term is summed over. This is called the Einstein summation convention. So is the term p(mu)*q(mu) equivalent to the scalar product of the vectors p and q (which can also be written as p.q).

There are of course also indices that should not be summed over. Such indices we call zero-dimensional. This is just a convention. To declare indices we use the statement:

```
Index name[={number|symbol}]
              [,othername[={number|symbol}]];
```

When the equals sign is used this indicates the specification of a dimension. Indices that are not followed by an equals sign get the dimension that is currently the default dimension (see also the section on vectors). The dimension can be either a number that is zero or positive (zero indicates that the summation convention doesn't apply for this index) or it can be any symbol with the exception of the symbol i_. The symbol must have been declared before.

The most important use of the dimension of an index is the built in rule that a Kronecker delta with two the same indices is replaced by the dimension of this index if this index has a nonzero dimension. So when mu is 4-dimensional d_(mu,mu) will be replaced by 4 and when nu is n-dimensional d_(nu,nu) will be replaced by n. If ro is zero dimensional the expression d_(ro,ro) is left untouched.

In addition to the symbolic indices there is a number of fixed indices with a numeric value. The values of these indices runs from zero to an installation dependent number (usually 127). If the user likes a different maximum value he should consult the chapter on the setup parameters. The numeric indices are all assumed to have dimension zero, so no summation is applied to

them. This means that they can be used for vector components. It is therefore perfectly legal to use:

```
V    p,q,r;
L    F=p(1)*q(1)*r(1)+p(2)*q(2)*r(2);
```

When two numeric indices occur inside the same Kronecker delta a value is substituted for this delta. Normally this value is one when the two indices are the same and zero when they are different. The value for the diagonal elements can be changed with the 'FixIndex' statement:

```
Fi[xIndex] number:value [,number:value];
```

This command assigns to d_(number,number) the given value. This value must fit in a single FORM word. This means that this value can at least be in the range -32768 to +32767. For more details on the size of a FORM word one should consult the installation manual.

In the case of summable indices the use of three times the same index in the same term would cause problems. FORM will execute the contraction for the first pair it encounters, after which the third index is left. In the case of four or more indices the pairing for the contractions depends on the order in which the parts of the term is processed, so on the userlevel the result may be quasi random. Nothing can be done about this as the user should guard against such ambiguous notation.

6.5 Functions

There are two types of functions: **commuting functions** which commute automatically with all other objects, and **noncommuting functions** which do not necessarily commute with other noncommuting functions. An object is declared to be a commuting

function with the 'cfunction' command. Of this command the first two characters are mandatory, the others optional. An object is declared to be a noncommuting function with the 'function' command. Here only the f is mandatory. The declaration of a function knows one option. This option concerns the complexity properties of the function. It is indicated by a # following the name, after which one of the characters R, I, C specifies whether the function is real, imaginary or complex. The declaration that a function is real is unnecessary as real is the default property. Example:

```
cf   fa,fb,fc;
f    ga,gb,gc#c;
```

In this example the functions fa, fb, fc are commuting and the functions ga, gb and gc are not necessarily commuting. In addition the function gc is complex. More about functions and their conventions is explained in the chapter on 'Operations on functions'.

6.6 Sets

A set is a (non empty) collection of variables that should all be of the same type. This type can be symbols, vectors, indices or functions. A set has a name which can be used to refer to it, and this name may not coincide with any of the other names in the program. They are declared by giving the name of the set, followed by a colon, after which the elements of the set are listed. The first element determines the type of all the elements of the set. All elements must have been declared as variables before the set statement. There can be only one set per statement. Example:

```
s    xa xb xc xd ya x y;
i    mu nu ro;
```

```
set exxes: xa, xb, xc, xd;
set yyy: xc, xd, xb, ya;
set indi:  mu, nu, ro, 1, 2, 3;
set xandy: xa, ya;
```

We see here that a single symbol (xa) can belong to more than one set. Also the fixed indices (1, 2 and 3) can be elements of a set of indices and the numbers that can be powers can also be members of a set of symbols (usually -9999 to + 9999). If this can lead to a confusion FORM will give a warning and interpret the set as a set of symbols.

Sets can be used during wildcarding. When x is a symbol the notation x? indicates 'any symbol'. This is sometimes more than we want. In the case that we would like 'any symbol that belongs to set exxes' we would write x?exxes which is an unique notation as usually the question mark cannot be followed by a name. There should be no blank between the question mark and the name of the set. The object x?indi would result in a typemismatch error if x is a symbol and indi a set of indices.

This use of wildcards belonging to sets can be extended even more: The notation x?exxes?yyy means that x should belong to the set exxes, and its replacement should be the corresponding element of set yyy. At first this notation looks unnecessarily complicated. The statement

```
id   x?exxes?yyy = x;
```

should have the much simpler syntax

```
id   exxes = yyy;
```

This last notation cannot be maintained when the patterns are more complicated.

```
f    f,g;
id   x?exxes?yyy = 2*x^2;
id   x?exxes?yyy * f(x?yyy) = g(x-x^2);
```

In the last example the loose x must belong to set exxes while the x in the function f must belong to the set yyy. As the two occurences of x must match the same object it means that x must belong to the intersection of these two sets. The replacement of x will be by the element of yyy that has the same number inside yyy as the x has inside the set exxes, so xa doen't match, xb goes to xd, xc goes to xb and xd goes to ya. When the two sets exxes and yyy have a different number of elements the compiler will issue an error message.

When things become really complicated the sets can be used as kind of an array. They can be used with a fixed array index (running from 1 for the first element). When they have a symbolic argument (must be a symbol) they are either in the right hand side of an id-statement and the symbol must be replaced by a number by means of a wildcard substitution or in the left hand side and the symbol is automatically seen as a wildcard. The set must still follow the question mark of a wildcard:

```
s a1,a2,a3,b1,b2,b3,x,n;
f g1,g2,g3,g;
set aa:a1,a2,a3;
set bb:b1,b2,b3;
set gg:g1,g2,g3;

id  g(x?aa[n]) = gg[n](bb[n]) + bb[2]*n;
```

The n in the left hand side is automatically a symbol wildcard. x must match an element in aa and n takes its number. In the right hand side `gg[n]` becomes an array element when the n is substituted. The same holds for `bb[n]`. The element `bb[2]` is

immediately replaced by b2, so there is rarely profit by using this, unless the preprocessor had something to do with the construction of this quantity. As should be clear from the above: the array elements are indicated with straight braces.

Another use of sets is in the select option of the id statement. This is discussed in the chapter on substitutions.

6.7 Namelists

Sometimes it is necessary to see how FORM has interpreted a set of declarations. It can also be that declarations were made in an unlisted include file and that the user wants to know what variables have been defined. The lists of active variables can be printed with the statement

```
write names;
```

This statement sets a flag that causes the listing of all name tables and default properties that are active at the moment that the compiler has finished compiling the current module. This is just before the algebra processor takes over for the execution of the module –assuming that no error condition exists–. If the 'write names' is specified in a module that ends with a .global instruction the namelists will be printed at the end of each module, as printing the namelists will then be the default option. If one likes to switch this flag off this can be done with the statement

```
nwrite names;
```

which prohibits the printing of the namelists in the current module.

6.8 Dummy indices

Sometimes indices are to be summed over but due to the evaluation procedures in some terms there is the index mu and in other terms there is the index nu. There is a command to sum over indices in such a way that FORM recognizes that the exact name of the index is irrelevant. This is the 'sum' statement:

```
i  mu,nu;
f  f1,f2;
L  F=f1(mu)*f2(mu)+f1(nu)*f2(nu);
sum  mu;
sum  nu;
print;
.end
```

At first the expression contains two terms. After the summations FORM recognizes the terms as identical. In the output we see the term:

```
2*f1(N1_?)*f2(N1_?);
```

The `N1_?` are dummy indices. Currently FORM can print them but cannot read them when the output is offered as input to the compiler. The dimension of these dummy indices is the current default dimension as set with the last dimension statement. This may look like it is a restriction, but in practice it is possible to declare the default dimension to have one value in one module, take some sums, and do some more operations, and then give the default dimension another value in the next module.

The scheme that is used to renumber the indices in a term is quite involved but alas not perfect. To do a perfect job could consume enormous amounts of computer time when complicated products of Levi-Civita tensors are present.

6.9 Restrictions

The number of each type of variables is restricted. This restriction has a default value that depends on the installation and is usually 100 for each of the variables including the expressions. It is however possible to change these limits with a setup file. For more about this one should consult the chapter on the setup file. In addition there is a limit to the total number of elements for all sets together which can however be altered with the setup file.

Substitutions

The essence of a symbolic manipulation program is the ability to replace an object by other objects. We call this a substitution. The substitutions are divided in two classes: **identifications** and **operations**. The operations perform a specified action and are treated in the chapter on operations. An identification is a replacement of a specified pattern. Usually when we call something a substitution we mean such a replacement. One of the tasks of this chapter is to describe the patterns that can be specified.

The identification statement starts always with the keyword id[entify]. There may be a secondary keyword, depending on the pattern. Then there is a pattern which forms the left hand side of an equation, followed by an equals sign and a right hand side. If the given pattern can be located in a term it is taken out and the right hand side is substituted for it. Example:

```
id  x = a + b;
```

The pattern here is rather trival. The above identification will take any power of x and replace it by the same power of `a + b`. In other words: each occurrence of x is replaced by `a + b`. Patterns can be more complicated:

```
id  x*y^2*p(mu) = ((a+b)*q(mu)+(a-b)*r(mu))/2;
```

Here each occurrence of the pattern is replaced by the right hand side. When mu is an index with a dimension this will be at most once, as in a pattern of this kind only a free occurrence of a p with index mu can lead to a match.

Often we need a way to indicate a type of pattern, rather than a specific pattern. This is done via **wildcarding**. A wildcard is

a variable that occurs in the left hand side of an identification statement and of which the name is followed by a question mark. So the pattern `p(mu?)` means a p with any index. The index will be called mu during the replacement, regardless of whether there exists a variable mu already. mu must have been declared as an index before:

```
id  p(mu?) = (q(mu)+r(mu))/2;
```

Any occurrence of p with an index is replaced by the right hand side with the found index taking the place of the mu in the right hand side. There can be more than one wildcard in a pattern:

```
id  p(mu?)*p(nu?) = T(mu,nu) + d_(mu,nu)*p.p;
```

When the same wildcard occurs more than once in a pattern all its matches must be identical:

```
id  Q?(mu)*Q?(nu) = d_(mu,nu)*Q.Q/4;
```

The above pattern will match when there are two identical vectors, one with index mu and the other with index nu.

The following objects are allowed inside a pattern:

x
: The symbol x. Each power will give a replacement.

x^number
: An integer power of x (0 is forbidden). There will be as many replacements as possible. Example: x^3 fits twice in x^7.

x^n?
: This matches any (integer) power of x.

x?
Any symbol. A fixed power is also allowed.

x?^n?
Any symbol to any (integer) power. The difference with the 'any symbol' pattern is found in the number of right hand side insertions when a power of a symbol is encountered: In the 'any symbol' pattern there will be as many right hand side insertions as there are powers of the symbol, while the pattern with the wildcard power will never give more than a single right hand side insertion for which n takes the value of the power of the symbol.

p(mu)
The vector p with index mu. In this form p has to occur as a vector with an index. There will be no match with vectors of which the index has been contracted. In particular there will be no match with vectors inside a dotproduct. Powers are not allowed for vectors. If such a power is nevertheless specified the exponent function is invoked. The best way to achieve the equivalent of (p(mu))^n? is to replace p(mu) by a symbol x and then use the pattern x^n?.

p(mu?)
The vector p with any index.

p?(mu)
Any vector with the index mu.

p?(mu?)
Any vector with any index.

p.q
The dotproduct p.q. Any fixed power of a dotproduct or its

wildcard forms is also allowed as in the x^number pattern.

p.q? or q?.p
Any dotproduct that contains at least one occurrence of the vector p.

p?.q?
Any dotproduct. Note that which vector in the dotproduct will be assigned to p and which will be assigned to q is a function of internal ordering in FORM. This plays no role when the right hand side is symmetric between p and q. If such a symmetry does not exist there is an arbitrariness in the answer. The user is advised not to make programs that depend on the order in which FORM assigns these wildcards as this may be different in future versions or different implementations.

p?.p?
Any dotproduct that is the square of a vector.

p.q^n?
All the above forms of the dotproduct may have a wildcard power.

d_(mu,nu)
Patterns may contain Kronecker delta's.

d_(mu?,nu) or d_(nu,mu?)
Any Kronecker delta that contains at least one index nu. Note that mu can match one of the fixed numerical indices, but that currently one has no wildcard to pick out only those indices.

d_(mu?,nu?)

Any Kronecker delta. Again the internal ordering of FORM determines which index is match to mu and which index takes the place of nu. If the right hand side is not symmetric between mu and nu the result is a function of the order in which mu and nu were declared.

d_(mu?,mu?)

Any Kronecker delta with two the same indices. This index must have dimension zero, or the Kronecker delta would not occur in the term. Note that currently there is no way to wildcard the indices in such a way that only indices with the same dimension will match it.

p

In this form p must be the only object in the pattern. An automatic wildcarding will be assumed for the index and the substitution will be made in all occurrences of p as a vector with an index, of p inside dotproducts and of p contracted with indices inside the special functions e_ and g_. Occurrences in other functions should be handled according to the methods that are given in the chapter 'Operations on functions'. In this form each vector without an index on the right hand side gets the generated wildcard index attached to it:

```
id  p = (q+r)/2;
```

It is of course easy to produce meaningless results by making putting 'too many' vectors in the right hand side:

```
id  p = q*r;
```

Such an identification can easily lead to a messed up output.

When using this type of replacement the user ventures out on his own.

Sometimes one needs to refer to the induced wildcard. This can be done with a single question mark as in:

```
id  v = e_(p,q,r,?);
```

In this case a new dummy index is generated. If it makes it to the output it will be printed as Nnumber_? in which 'number' is a number that is as close to 1 as possible without coming into conflict with other generated dummy indices. This type of dummy index always has the dimension that is currently the default dimension. If the default dimension is changed the dimension of the dummy indices changes with it. Basically the default dimension is the dimension of the vectors inside dotproducts.

p?

The use is similar to the preceeding pattern, but here the vector is a wildcard.

More patterns are given in the chapter 'Operations on functions'.

The normal rules for the pattern matching are that a pattern is taken out as many times as possible, unless a wildcard power is present, in which case the pattern is matched only once. Only after all matches have been taken out are the right hand sides inserted. After this insertion there is no renewed attempt to match the same patterns again. When there is a need to deviate from the above rules one may specify a subkey. These subkeys are:

Once

Only a single match of the pattern is attempted. In partic-

ular the statement

```
id,Once,x = a + b;
```

will take only one power of x, so `x^8` will go to `x^7*(a+b)`. This is the default mode when there are wildcard powers of symbols or dotproducts in the pattern. In such a case only the subkeys 'Once' and 'Only' are allowed.

Only

The match must be exact. This means that all powers of the symbols and the dotproducts must match exactly with the powers of these symbols and dotproducts inside the term.

Multi

There is a single matching for which a multiplicity is determined. This means that the right hand side will be used only once but it may be taken to some power. This form can only occur when the pattern contains only symbols and/or dotproducts and there are no wildcard powers present. In that case multi is the default option. Note that in this case there is only a single wildcard assignment.

Many

This is the default option when a pattern contains objects that are neither symbols nor dotproducts. It may also not contain wildcard powers of symbols or dotproducts. With this option the matching is attempted till there are no more matches. Each match gives an insertion of the right hand side with its own wildcard assignments after all pattern matching has been completed.

Select

This option is a special form of 'only'. It needs in addition the names of one or more sets of symbols. The replacement will then be made if after the matching of the left hand side no elements of the mentioned sets are left, unless they occur inside function arguments. Example:

```
s  a, b, c, d;
set ab: a, b;
L  F = (a+b+c)^5;
id,select,ab, a = d;
```

Only a single 'a' will match if there are no elements of the set ab left after the match. This means that a*b∧4 will not give a match but a*c∧4 will. If more sets are specified the union of the sets is taken. In that case the names of the sets must be separated either by blanks or by comma's.

There will be many more subkeys in future versions to allow for great flexibility in the selection of the patterns to be acted upon.

Patterns have to behave according to some fixed rules. One of these rules is that a pattern may never develop a coefficient that is not equal to one. This means that

```
id  2*x = a + b;
```

is illegal and will result in a compile time error. The same holds for the following pattern:

```
id  d_(mu,mu) = 4;
```

if mu has a dimension that is not set to zero. During the compilation the left hand side is normalized and part of the normalization

is the contraction of indices. So if the dimension of mu is 4 the statement would have just 4 on the left hand side and would be illegal. If the dimension of mu would be 'n' the left hand side would become n and the statement would be perfectly legal, even though it might not be what the user had in mind.

Each symbol or dotproduct may only occur once outside function arguments. The following pattern is illegal:

```
id  x^2*x^n? = y^2 * a^n;
```

It is usually not very difficult to work around this type of restriction.

Sometimes one would like to take out the left hand sides of more than a single substitution before the right hand sides are inserted. This is for example the case in the following two dimensional rotation:

```
id    p = costheta*p + sintheta*q;
idold q = costheta*q - sintheta*p;
```

It would be rather destructive to the idea of the rotation if the second substitution would act on the results of the first. The id statement and the idold statements that follow it define a group of substitutions. In thisgroup all left hand sides are taken out first (in order) and after this has been done the right hand sides are inserted. Next is the statement after the last idold of the group.

For those people that are used to Schoonschip the idold statement has the alias al[so]. As it turns out people prefer to use it this way so in most of the examples the idold statement is used in the guise of the 'al' statement. They are completely identical.

Operations on functions

Because of their arguments the rules for operations on functions are somewhat different from the normal operational rules. There are basically two types of operations: one is a class of regular substitutions with potentially rather complicated wildcards, while the other concerns the properties of functions. We will first deal with the properties of functions. There are several commands for defining properties. One set is executed at the moment it is specified while the other set concerns more durable properties. Of course durable properties will have a tendency to cost more computer resources, as they will have to be checked everytime that anything is done with the function. Currently the following property functions have been implemented:

Symmetrize Name Fields

This statement is executed only when specified. The name is the name of the function and the fields refer to the arguments. Each field is treated as a single entry for purposes of symmetry. The simplest field is a single digit, indicating the number of a single argument. So

```
Symmetrize Fun 1,2,4;
```

defines that the function Fun is to be symmetrized in its arguments 1, 2 and 4. If an occurrence of Fun is found with fewer than four arguments the statement is ignored. When the function name is specified as Fun:number the number indicates that only occurrences with the given number of arguments should be affected. The notation Fun: indicates

that if the number of arguments is too small the excess number of specified fields will be ignored. So if Fun were to have three arguments and we had specified Fun: 1,2,4 only the first two arguments would be symmetrized.

The second type of argument fields concerns a group of arguments. It is specified with the numbers of the elements of the group enclosed in parentheses:

```
Symmetrize Fun (1,2),(4,5),(7,8);
```

In this case the arguments 1 and 2 are considered together. This means that if 1 is exchanged with 4, 2 will be exchanged with 5 and vise versa. When groups are used all fields must be groups of the same length or an error message is given. Note also that if an occurrence of Fun is found with seven arguments it will not be treated and the seventh argument will never be part of symmetrization even when the statement would have been

```
Symmetrize Fun: (1,2),(4,5),(7,8);
```

to indicate that the number of arguments is not important. There is one severe restriction: the elements of an argument field must be adjacent. This means that

```
Symmetrize Fun: (1,3),(4,5),(7,8);
```

would be illegal. Also all fields must have the same number of elements.

The ordering of the arguments is according to the orderin the namelists of FORM. The order in the namelists is determined by the order of declaration of the variables.

When no argument fields are specified the function is assumed to be symmetric in all its individual arguments.

Antisymmetrize Name Fields
This command is fully equal to the symmetrize statement with the exception of the minus sign that is added to the coefficient for each exchange of a pair of argument fields. If two argument fields are identical the function is replaced by zero.

The more durable equivalents have not yet been installed. Their names would be 'symmetric' and 'antisymmetric'.

The regular replacements of functions are rather simple. When noncommuting functions are involved the order of the functions is important unless a special property statement (not yet implemented) was used to declare commutation properties. Also special keywords (currently not implemented) may relax the rules about ordering of functions.

8.1 Wildcarding

With respect to substitutions of functions there are several types of wildcarding. The simplest case is a wildcard parameter that has to match a single parameter that is a single object of the same type. This is the case in the following example:

```
s    x;
i    mu,nu,rho;
v    p,q;
f    f,g;
id   f(p?) = p.q;
id   f(mu?) = g(mu);
id   f(x,mu?) = q(mu);
```

```
id   f(x?,mu?,nu?,p?,rho) = x * p(rho) * d_(mu,nu);
```

The last id statement shows that there can be more than one wildcard per function. Wildcards may be repeated in the left hand side. This means that each occurrence has to match with the same object:

```
id   g(x,mu?,p,mu?) = f(x,p);
```

will match `g(x,nu,p,nu)` but it won't match `g(x,nu,p,rho)`. It is of course possible to have more functions or functions and other objects at the left hand side of the id statement. Again the rules are that wildcards that are identical at the left hand side should match with identical objects.

The second type of wildcard is a wildcard for the function itself:

```
id   f?(x) = f(0,x);
```

The above causes each function of x to obtain an extra argument.

The third type of wildcard is a little bit intertwined with the first type. Wildcard arguments can match a whole composite argument:

```
id   f(x?) = x^2;
```

will match with `f(a+b)` and the result will be `(a+b)^2`. A symbol wildcard will only match whole arguments if the argument doesn't contain loose vectors or indices. A vector wildcard can match a whole argument if the nature of the argument is vectorlike. This means that each term in the argument contains a single loose vector. An index wildcard will match a whole argument if the argument is either a single index or vectorlike. In the last case the argument is considered to be a single index that has

been contracted with a vector expression. In such a case it is also clear that the function is linear in its argument. This linearity is exploited.

```
id   f(mu?) = g(mu);
```

When this statement matches `f(2*p+q)` this property makes the result to be `2*g(p)+g(q)`. If this is not desired one should use the vector wildcard, rather than the index wildcard.

The fourth type of wildcard is very special. It involves groups of arguments of functions. In the statement

```
id   f(???) = g(1,...);
```

the ??? will match any argument field of f. In the right hand side the three dots refer to this match. This means that `f(a,b)` will be replaced by `g(1,a,b)`. There are ten of these 'argument field' wildcards. They are referred to by the number of question marks and dots that are used. So the following is legal:

```
id   f(??)*g(??)*h(x,?) = h(x,.,..);
```

This will match when there are an f and a g with the same argument fields and an h with an argument field that starts with an argument that is a single x. The single period in the right hand side will be replaced by whatever the single question mark matches (it could also be no argument at all). The double dot will be replaced by the match of the double question mark in the left hand side.

There is one severe restriction. Per function one may use only one 'argument field' wildcard. There may be any number of other arguments though.

Currently no provisions have been taken for wildcard matches of the type

```
id   f(x?+a) = x * g(2*x+a);
```

Such wildcarding could be very costly in time. The complexity of implementing this in a very general (and recursive) way is very great.

Restrictions:

The above wildcardings can be applied to all functions with the exception of the functions e_, g_, d_. With these functions only wildcards of the first type can be used.

Repeat

In some symbolic manipulation languages the specification of a substitution means that this substitution is applied until it doesn't give any further matches. In such a case the replacement

```
id    x^2 = x + 1;
```

would take any power of x and reduce it down until there are only terms with a single power of x or no power of x (not counting negative powers). Another approach is to apply the substitution once and allow the user to specify a more frequent use. The more frequent use has to be specified carefully in some cases. It can cause unpleasant surprises when it is wrongly estimated.

The solution in FORM is to have each individual statement applied only once, –in the case of the keyword many there is also only one cycle of matching and then inserting– but allow a repeated use when a special instruction is provided. The syntax is:

```
repeat;

 any number of id statements, idold statements
 or operations.
 All statements must be part of the same module.

endrepeat;
```

The statements between the repeat and the endrepeat statements are applied as a group until they cause no further changes.

This provides much more flexibility then either of the above models for substitutions. There are of course some caveats. The statements

```
repeat;
id   x = x + 1;
endrepeat;
```

will cause an infinite loop of substitutions. How execution will come to a halt is installation dependend as not all implementations allow tests for stack overflow.

Repeat loops may be nested to 10 levels. The only restriction is that all statements in a loop belong to the same module. This means that no .sort, .store, .global or .end instruction can occur in the range of a repeat loop.

Partial Fractioning

When a formula that contains composite denominators has to be integrated a standard method is to 'split the fractions' which is also called partial fractioning. The basic identities for such an operation are:

```
1/(x+a)/(x+b) = (1/(x+a)-1/(x+b))/(b-a);
(x+a)/(x+b)   = 1 - (b-a)/(x+b);
```

This partial fractioning is part of a package of polynomial operations that has not yet been written. To execute the partial fractioning with single symbols one can use the ratio statement. It operates on symbols, so normally a single symbol xb would have to represent the composite denominator like (x+b). The name-convention with the `[` and the `]` was designed to avoid such cryptic notation. It is usually very convenient to write `[x+b]` for (x+b). The ratio command should be given in the following way:

```
ratio,xa,xb,ba;
```

It expects three parameters that must have been declared as symbols. The third parameter is supposed to stand for the difference between the second parameter and the first parameter: `ba=xb-xa`. In answer to this command FORM will look for combinations of powers of xa and xb with at least one of them in the denominator. If such a combination is found it is rewritten in such a way that xa and xb don't occur in the same term any more. The case with both powers positive is usually rather simple and doesn't need any special algorithms. In principle the case with `xa^n/xb^m` can be solved by replacing xa by xb-ba, so the ratio command uses

for the case that n > m a slightly different algorithm in which the terms that have no xb in the denominator are still expressed in terms of xa. This could be called a 'minimal' operation. Example:

```
s   [x+a],[x+b],[b-a];
l   F1 = 1/[x+a]^2/[x+b];
l   F2 = [x+b]^2*[x+a]^-1;
ratio,[x+a],[x+b],[b-a];
```

This gives the formulae:

```
F1 =
    - [x+a]^-1*[b-a]^-2 + [x+a]^-2*[b-a]^-1
    + [x+b]^-1*[b-a]^-2;

F2 =
   [x+a]^-1*[b-a]^2 + [x+b] + [b-a];
```

The second formula could also be obtained by a pair of substitutions:

```
s   [x+a],[x+b],[b-a];
l   F2 = [x+b]^2*[x+a]^-1;
id  [x+b] = [x+a] + [b-a];
id  [x+a] = [x+b] - [b-a];
```

This would require much more work as there would be more terms generated than is needed. To avoid the ratio command for the first formula would be much more complicated.

A typical example of the use of ratio is given below:

```
s   [x+a],[x+b],[b-a],dx,x;
s   [ln(x+a)],[ln(x+b)],n,m,da;
set [x+]: [x+a] [x+b];
*
```

```
L   F = dx/[x+a]^4/[x+b]^5;
*
*   Partial fractioning:
*
ratio,[x+a],[x+b],[b-a];
*
*   Now do the logarithms first:
*
id,only,dx/[x+a] = [ln(x+a)];
id,only,dx/[x+b] = [ln(x+b)];
*
*   Now we can do the other integrals.
*   These wildcards work for linear
*   polynomials and all their powers.
*   They must be in set [x+].
*
id  dx*x?[x+]^n? = x^(n+1)/(n+1);
b   [b-a];
print;
.end

  F =
      + [b-a]^-4 * (  - 1/4*[x+b]^-4 )
      + [b-a]^-5 * (  - 1/3*[x+a]^-3
                      - 4/3*[x+b]^-3 )
      + [b-a]^-6 * ( 5/2*[x+a]^-2
                      - 5*[x+b]^-2 )
      + [b-a]^-7 * (  - 15*[x+a]^-1
                      - 20*[x+b]^-1 )
      + [b-a]^-8 * (  - 35*[ln(x+a)]
                      + 35*[ln(x+b)] )
```

In the above the logarithms are introduced as separate symbols which is a common technique when the expressions become big while there are only few different arguments for the logarithm. The implementation of a dedicated logarithm function is planned for some future version.

Special functions

Several special functions have been implemented in FORM. They are usually functions that are either hard to immitate or costly in execution time when the user defines them by himself. These functions are:

fac_

This is the factorial function. Currently it should have only a single argument. When this argument is a positive integer the factorial will be substituted. The first 50 factorials are kept in memory (unless they have never been used). The higher factorials are evaluated, starting from 50. This may be altered in the future.

theta_

This is a step function. When theta_ has a single argument,and the argument is just a (rational) number FORM will replace the function by 0 when the argument is negative and by one otherwise. No attempt is made to guess whether the argument can be negative in the same way a human would do with for instance theta_(x^2) when x is real. When there are two arguments the arguments are compared with each other and when this comparison indicates that the arguments should be exchanged, assuming that the function were symmetric, the function is replaced by zero. If the arguments are in order or identical the function is replaced by one.

delta_

This is a delta function. It has two varieties .In the first variety it should have two arguments. When the two arguments are identical the whole function is replaced by 1. When the two arguments differ and the are both (rational) numbers the function is replaced by zero. When the arguments differ and at least one of the arguments contains variables the function is left untouched. When there is only one argument the function is replaced by 1 if the argument is zero, by zero if the argument is a number that is unequal to zero. In all other cases it is left untouched. Note that delta_(a,b) gives the same result as delta_(a-b).

The count instruction

Often one has problems in which a number of terms can be eliminated on the basis of power counting. One of the solutions could be to introduce a new symbol with a limited power range and then replace all relevant objects in such a way that the power counting corresponds to the powers of this new symbol. This can be quite cumbersome. There exists a special statement for power counting purposes. It is called the count statement. Its syntax is:

```
count minimum,object,value[,object,value];
```

'minimum' is the minimum value of the count that is needed if the term is to survive the power counting. When the combined count is less than 'minimum' the term is removed. After the minimum there follows an arbitrarily long list of objects and the value that is used for the counting. Both 'minimum' and these values should be integers that fit inside a short integer (from -32768 to +32767 on most systems). The objects can be of several types. They must have been declared before. The allowed types are:

Symbol
: Object is the name of a symbol. When the count is made value is multiplied by the power of this symbol and added to the count. Only symbols that don't occur inside function arguments are considered.

Function
: Object is the name of a function. For each occurrence of this function outside function arguments value is added to the count.

Dotproduct

Object is a dotproduct. Dotproducts are handled in the same way as the symbols.

Vector

Object is the name of a vector. In this case there exist possible ambiguities. There is also a need to have some control over which occurrences of the vector are taken into account. There are again four cases:

Vector	Loose vectors with indices are taken into account.
Dotproduct	Vectors inside dotproducts are taken into account.
Function	Vectors inside the special linear functions e_ and g_ are taken into account.
Sets	Vectors that are the argument of a function that is a member of the specified set are taken into account. This assumes that thefunction is linear in the vector.

In order to have some control over the various cases there is the possibility to add some flags after the name of the vector. This is done by typing a '+' after the name of the vector, followed by one or more characters. 'v' stands for vector, 'f' for function, 'd' for dotproduct and '?setname' for the set option. The set option is always the last option and there can only be one set specified. When there is no '+' following the name of the vector the result is identical to vectorname+vfd.

In principle there can be an interference between the counting of vectors and the counting of dotproducts. The rule is that if an object occurs more than once in the list it is counted more than once, so

```
count 0 p 1 p.q -1;
```

gives for the dotproduct p.q a count of 0 and for the dotproduct p.k a count of +1 (assuming that p,k and q are vectors). The dotproduct p.k^5 would give a count of 5 and p.p would give 2. The term p(alfa)*p.q would give a count of 1.

There is a variation to the count statement which is the count function. It can occur in the condition field of an if statement. Its parameters are identical to the arguments of the count statement with one exception: the first parameter (minimum) is missing. This function returns the value of the count. Actually the count statement

```
count minval,arguments;
```

is identical to:

```
if ( count(arguments) < minval );
    discard;
endif;
```

in which the discard statement removes the terms that have a count that is too small.

Output

As the primary object of a program is to produce some output there must be some ways to control this output. On the whole this interface is not as fancy as in some other symbolic programs, because FORM is primarily intended as a package to manipulate giant formulae, rather than being a desktop program for presenting results in an artistic way. There are several instructions available in FORM:

Print

The print statement tells FORM to print its results at the end of the module. If no expressions are specified all files will be printed. If expressions are specified only those expressions will be printed. Only active expressions can be printed. The statement:

```
print[];
```

causes the printout to take a special form. Of each expression only the external part outside the brackets is printed, while of the part inside the brackets only the length is given. As this is a print statement without the names of expressions all expressions will be printed. The statement:

```
print name1,name2[],name3;
```

causes the printout of the expressions name1, name2 and name3. The expression name2 is printed in the special notation that is explained above with the `print[]` statement.

Nprint

This statement causes the exclusion of expressions from the list of expressions to be printed. Just the single nprint statement removes all expressions from this list. If names are specified only those expressions are taken from the list. This can be used in the following way:

```
print;
nprint name5;
```

These instructions cause the printing of all expressions with the exception of name5.

There can be several option flags in the print statements. These flags can be put between the names of the expressions or as parameters at the end of the statement when no expressions are specified. These flags are:

+f
: This option causes the expressions to be printed only in the log file if there is one. If there is no log file this flag is ignored and printing is as usual.

-f
: This turns the above flag off again.

+s
: The expression(s) will be printed with each term starting at a new line.

-s
: This turns of the flag for single terms per line.

Example:

```
print a +f b +s c -f d;
```

The expression a is printed as usual (screen plus log file). b is printed only in the log file. c is printed only in the log file with one term per line only and d is printed on the screen and in the log file with a single term per line.

Print statements hold till the next .sort instruction. This means that for each print statement the requested expressions are printed only once. If the user wants to print the expression(s) also at the end of the next module he should issue another print statement in that module.

Bracket

This statement has arguments that indicate which object should be presented outside brackets. Only the initial b of the statementis mandatory, the other characters are optional. When the statement

```
b   x;
```

is given all powers of x will be taken outside parentheses. With vectors the situation is more confusing:

```
b   p(mu),q;
```

will put p(mu) and q with any index outside parentheses. This statement will not affect dotproducts. With respect to bracketing dotproducts should be handled as symbols, ie each of them should be specified separately.

Functions can be put outside parentheses by specifying their names. This may cause some problems with noncommuting functions, so the rule is that if a noncommuting function is

taken outside parentheses all noncommuting function to the left of it are also taken outside the parentheses.

There may be only one bracket statement per module. If there are more bracket statements only the last one counts.

Format

The format statement is used to either indicate the number of columns that the output is allowed to occupy, or to tell FORM that the output must be in a form that can be read by a fortran compiler. The first command is given by:

```
format number;
```

in which 'number' is a number of at most 255. The fortran output is forced with:

```
format fortran;
```

When output is given in the fortran mode there is still a little editing work left for the user. The number of continuation cards has been left unrestricted, even though all brackets have been split up in pieces of at most about 10 lines. If for instance the expression is called F the user must supply lines that read F=F between the lines of the output. There is a reason why FORM doesn't do this yet. The expression could just as easily have had the name F(a,b,p) or something like it. In that case inserting lines like F(a,b,p)=F(a,b,p) would give nonsense. With a good editor the job for the user should not be very great.

Preprocessor instructions

FORM is equipped with a preprocessor that can prepare the input for its compiler. It is the preprocessor that defines **modules** that are translated by the compiler and executed immediately after the translation. This makes FORM an interpreter on the level of entire modules, while inside it has the benefit of compiled code.

There are several classes of preprocessor instructions. They are easily recognizeable as their first character is always a special one. The modules are always terminated by a statement that starts with a period. These statements are:

.sort
: Executes the module, sorts its results, prints them when requested and prepares for further processing.

.end
: Executes the module, sorts its results, prints them when requested and terminates the program.

.store
: Executes the module, sorts its results, prints them when requested, stores global expression so that they may be used later in substitutions and removes all local variables. The next module will only know the global definitions.

.clear
: Executes the module, sorts the results, prints them when requested and makes a soft restart of FORM. The commandtail isn't read again, and neither is the setup file. The reading of the input continues at the next line in the input

file and the preprocessor symbols are kept (the statement could be inside a preprocessor variable!). All loaded procedures are also kept (same reason). All stored expressions are lost. The errorflags are reset. This statement allows the consecutive execution of independent programs unless there is a severe error condition in one of the early parts of the program (like a preprocessor error).

.global
The definitions and declarations in the module are made global. They cannot be removed any more in the current program except for by a .clear instruction.

The absence of a .end is not a fatal error. A warning will be issued and a .end will be inferred. Any other (erroneous) statement that starts with a . will cause a preprocessor error. Preprocessor errors are immediately fatal and no further compilation will be attempted. The reason for this may become clear in the sequel.

The preprocessor knows a (installation dependent) number of preprocessor variables. The name of these variables can consist of any alphanumeric character string of **at most 10 characters**. They are recognizeable as they are enclosed inside single quotes. When the preprocessor finds a quote in the text it will look for the matching quote and consider the text in between as a variable. If this word doesn't correspond to one of its variables an error message will be issued. When there is a match the whole string, including the quotes will be replaced by the character string that the variable stands for. If this variable again contains quotes they are treated as before. The main restriction on the replacement string is that the sum of the length of all replacement strings that are defined at any given moment must be less than a fixed installation dependent number. When a preprocessor variable is

defined its name is not enclosed between the single quotes.

The largest class of preprocessor instructions starts with the character # which should either be in column 1 or preceeded by whitespace. This includes:

#include filename
: The reading of the current input file is interrupted and input is taken from the indicated file. When the end of that file is reached the reading continues from the original file. Such an include file may again contain #include instructions. The maximum level of nesting of these files shall not exceed an installation dependent number (see PreLevels in the chapter on the setup file.

#-
: Stop listing the input. Normally FORM will send a copy of the input to the screen while error messages will be put after the statement they correspond to. After the #- the input is not echoed. This feature is often employed to include files that are used frequently.

#+
: Resume listing the input.

#define Name "string"
: This defines a preprocessor variable with the given name. Whenever it is encountered in the text it will be replaced by string. String is any string of characters that is enclosed in the double quotes, including single quotes, linefeeds etc. So a preprocessor variable that is defined in this way can contain complete statements.

#do name = value1, value2 [**,value3**]
: The code following till the first '#enddo name' will be exe-

cuted repeatedly, while the first time name takes the value value1, the second time value1+value3 etc till name is larger than value2. The default value of value3 is 1. The objects value1 etc may contain simple arithmatic expressions but they have to evaluate to numbers that fit inside a word. Operations that are allowed are ∗, /, +, -, % (modulus: a%b is a mod b) and nesting of parentheses. If value1 is larger than value2 the contents of the loop are skipped. This statement is fully equivalent to the fortran do loop: 'do .. name = value1,value2,value3' Here the use of curly brackets (see later in this chapter) to force the evaluation of the values is not needed: It is done automatically. It is not even allowed to start the first value with a curly bracket, because this would indicate a 'listed loop' (see below).

#enddo

Ends the innermost of the currently active loops.

#do name = {string1|string2|.....|stringn}

The statements till the first '#enddo name' are executed first with the substitution name = string1, then with the substitution name = string2 etc. These strings may contain any characters that will result in further proper processing. This instruction is not limited to a single line, as reading continues till the matching } is encountered. When any of the strings contain curly brackets these brackets must be properly matched inside each string separately. This is called a 'listed loop'.

#procedure name(argument1,argument2,...,argumentn)

Declaration of a procedure. For more information one shoul dconsult the chapter on procedures.

#endprocedure

End of a procedure. This statement is only used to indicate the end of a procedure that is defined at the beginning of a program in which it is used. In the procedures that are defined externally in a file this statement is ignored.

#call name {string1|string2|.....|stringn}

Call of a procedure. The strings are the arguments that are substituted into the parameters in the procedure definition. For more information one should consult the chapter on procedures.

#if expression condition expression

The statements after this #if instruction are executed depending on whether the condition is true. The expressions must be numerical expressions that are evaluated by the preprocessor. The conditioncode can be

- `==` The condition is true when the expressions have the same value.
- `!=` The condition is true when the expressions have different values.
- `>` The condition is true when expression 1 is greater then expression 2.
- `>=` The condition is true when expression 1 is greater then or equal to expression 2.
- `<` The condition is true when expression 1 is less then expression 2.
- `<=` The condition is true when expression 1 is less then or equal to expression 2.

When there is only a single expression the following statements are executed when the expression is equal to a number that is not equal to zero. Together with the #if instruction

there are the #else and the #endif instructions. They are used in the same way as in any higher language.

#else
See the #if instruction.

#endif
See the #if instruction.

The loops are preprocessor loops i.e., they result in the repeated generation of code to be offered to the compiler. The string substitution is however very powerful and would not be possible inside the compiler. The compiler loops are of a greatly different nature and can be looked up in the compiler manual under the 'repeat' statement and the chapter on the if statement. There are also other varieties.

FORM can read two notations that are in use for exponents. The preprocessor will translate the combination ** into ^ internally, so both x**2 and x^2 are acceptable. Also both p1.p2 and p1$p2 will be interpreted as the vector product of p1 and p2 (the p1$p2 notation can be used in a fortran program).

Spaces are relevant in FORM when they separate words. They are then replaced by a comma internally. When they are adjacent to an operator or a bracket they are ignored. Tabs are fully equivalent to blanks for processing purposes. The character \ followed by a newline character makes the current line connect to the next line without the preprocessor trying to look at the first character of that next line. This is not a very useful feature.

All lines that start with a * are taken to be comments and are ignored after they have been through the preprocessor. Statements are supposed to end with a semicolon. A carriage return is interpreted as a single blank, so it is seen as a separator of words. When this is the opposite of what the user wants he can place an

'escape character' at the end of the line. The escape character is the backslash. When an expression is continued on the next line the continuation can start in column 1, unless the first character of this continuation is a $*$, because in that case the line would be interpreted as commentary. All characters that follow a semicolon until the end of the line are interpreted as commentary. There is no such thing as more than one statement on one line. Preprocessor statements are exempted from the rule that statements should be terminated with a semicolon.

It is perfectly legal to start a statement, continue it with the contents of one or more include files and terminate it after that. It may also happen that the statement is terminated inside the include file. Note however that a #do and the corresponding #enddo must lie inside the same file, unless the user wants to risk a crash.

Once the end of a statement is found the statement is immediately compiled and error messages are printed if there are any.

Sometimes it is necessary to evaluate a numeric expression in the preprocessor. For this the preprocessor is equipped with a little calculator. It can be activated by placing the expression to be evaluated between curly brackets. If the expression is fully numeric (no symbols etc) and each intermediate result fits inside a short integer the whole expression will be replaced by the character string that is formed by the numeric answer. Example:

```
G     f0 = 1;
G     f1 = 1;
.store
#do i = 2,10
G     f'i' = f{'i'-1}+f{'i'-2};
print;
.store
#enddo
```

```
.end
```

This is a (not so very smart) way to compute Fibonacci numbers. When 'i' is 2 the defining line becomes 'G f2 = f1 + f0;'. The expression to be evaluated may contain additions(+), multiplications(∗), subtractions(-), divisions(/), modulus(%) and brackets. The division is an integer division, so 5/2 will result in 2.

Summations

One way to program is series expansion is via the preprocessor loop facility. Although this is the most versatile method it is not always practical. The results of the preprocessor are fed to the compiler, so the summations in the preprocessor must necessarily fit in the compiler buffers. In addition they require compilation time. To avoid this there is a run time summation function named sum_. This function is treated as any other function, until its arguments fulfil a set of requirements:

- The first argument must be a single symbol. This symbol can be any symbol that was previously declared with the exception of the symbol i_. When the first argument is not a single symbol the compiler will issue an error message, because it must be known at compiletime which variable is the summation variable.
- The second argument is a number that fits inside a single word in FORM. This means usually that its value must be in the range -32768 to +32767. This value will be interpreted as the starting value of the summation variable.
- The third argument is like the second argument. It is used as a final value for the summation variable.
- There is an additional argument that is at the same time the second argument from the end and it fulfils the same requirement as the previous two arguments. It is used as an increment for the summation variable. If this argument is absent the increment is taken equal to one. When the increment is such that the starting value is already beyond the final value the sum is replaced by zero.

- The last argument can be anything. In particular it can depend on the summation variable. This argument is used as object to be summed over. For each value of the summation variable this value is sumstituted in this argument and the full argument is added to the sum.
- There are no arguments in addition to the ones mentioned above.

Under the above conditions the sum is evaluated and replaced by the results of this evaluation. Example:

```
s   x,i;
L   F1 = sum_(i,1,10,-(-x)^i/i);
```

The above is an expansion of the function ln(1+x) to 10 terms in x. Multiple sums can also be executed:

```
s   a,b,c,i,j,k;
L   F2 = sum_(i,1,3,a^i*
          sum_(j,1,i,b^j*
          sum_(k,1,j,c^k)));
```

When one or more of the arguments are not according to the required conditions the whole sum function is left as is. It may then be waiting for some substitution that makes evaluation possible. If such a substitution is never made the sum_ function will end up in the output eventually.

Note that this can be used to make more complicated sums:

```
s   a,b,c,i,j,k;
L   F2 = sum_(i,1,3,a^i*
          sum_(j,1,2*i-1,b^j*
          sum_(k,i+1,i+j+1,c^k)));
```

At first the inner sums won't be executed. The substitution of the value of i allows FORM to evaluate the third argument of the second sum, after which it can be evaluated after all. Eventually the second and the third arguments of the last sum are put in order and this sum can also be evaluated.

Of course sum_ has its limitations. When the summable object contains composite objects (like the (-x) in the first sum) that are taken to a (large) power the evaluation of this power may take much space in the inner buffers. If this causes an overflow of FORM's workspace the user can addapt the value of the variable 'WorkSpace' in the setup file (see the chapter on the setup parameters).

One should never try to give the summation variable the same name as an already existing wildcard variable:

```
id  x^i?*y^j? = i*x^(i+1)*sum_(i,1,j,z^i);
```

will result in an error message from the compiler. The use of the variable 'i' is ambiguous here. If the compiler would let this pass the term `x^2*y^3` would be replaced by

```
2*x^3*sum_(2,1,3,z^2)
```

and the sum would never be executed.

There is a second summation function called sump_. This function works like the regular function sum_ except for that the last argument is not the i-th element of the sum, but the quotient of the i-th element and the (i-1)-th element. The first element of the sum is normalized to one. This function can be useful at times:

```
L   F = sump_(i,0,10,x/i);
```

is the expansion of the exponent of x to ten powers in x.

Levi-Civita tensors

When doing either mathematics or vectoralgebra there may be a need for external products. A very simple example is the cross product between two three vectors. Such products can be written rather conveniently with the use of Levi-Civita tensors. A Levi-Civita tensor is a tensor that is antisymmetric in all its indices. It has as many indices as the dimension of the space that the indices refer to. In FORM the Levi-Civita tensor is written as e_(....indices...) and when the indices run from 1 to n e_ is defined by

```
e_(1,2,...,n) = 1
```

The antisymmetric properties fix all other values. In metrics that define upper and lower indices one may have to be more careful, but FORM doesn't know about upper and lower indices.The same conventions about contractions of indices with indices of vectors apply as in the rest of FORM. This means that

```
V    p1,p2,p3,p4;
L    F=e_(p1,p2,p3,p4);
```

defines a pseudo scalar object.

Levi-Civita tensors can also be used to define determinants. So the determinant of the 4x4 matrix A is defined by:

```
i    m1,m2,m3,m4;
cf   A;
L    F=e_(m1,m2,m3,m4)*e_(1,2,3,4)
      *A(m1,1)*A(m2,2)*A(m3,3)*A(m4,4);
```

The above determinant can be made more explicit with a special command. The product of a pair of Levi-Civita tensors can be rewritten into a series of terms that contain only Kronecker delta's d_. Thisreduction can be obtained with the 'contract' statement.

Often the contraction of Levi-Civita tensors should be controled. This is the case when there is more than a single pair of them. In the two dimensional example below it makes a big difference how the tensors are combined to form pairs:

```
i    m1,m2,m3,m4,m5,m6,m7;
L    F=e_(m1,m2)*e_(m1,m3)*e_(m4,m5)*e_(m6,m7);
```

One combination will give two terms, and the other two combinations lead to four terms. The results are equal but it is rather complicated to prove this without the original expression in terms of Levi-Civita tensors. When the 'contract' statement is invoked in its simplest form the pair that gives the smallest number of terms after contraction is combined and contracted. All other Levi-Civita tensors in the term are left as they are. In case of a tie the first pair involved in the tie will be taken. Successive application of 'contract' can remove all tensors (unless there is one left) and the expression that is generated this way will usually be the shortest.

To give the user some control over how many and which tensors are contracted the 'contract' statement has several variations:

Contract #

When # is a number contractions are executed till there are # or #+1 tensors left. A common example is the statement 'contract 0' which will contract as many pairs as possible.

Contract:#

The number here indicates the number of indices in a Levi-Civita tensor. Only tensors with the specified number of indices are contracted. This command can be used when there

are Levi-Civita tensors in several vectorspaces that have different dimensions. As with the regular contract statement only a single pair will be contracted.

Contract:#,#

The first argument refers to the number of arguments of the Levi-Civita tensors. The second argument specifies how many of those tensors must be kept.

In the 'Contract:#' statement we encounter an annoying problem. When there are tensors with different numbers of arguments one needs great care about the dimension of the indices. As a general rule FORM will consider an index that is common between two Levi-Civita tensors as contractible only when this index has the same dimension as the number of the arguments in both tensors. This means that more terms will be generated if one is sloppy about the notation and the result will also be different: the d_(mu,mu) term will be replaced by the dimension of the index, rather then by the number of arguments of the tensor.

When making substitutions on Levi-Civita tensors their antisymmetric properties are taken into account. With complicated wildcard assignments of wildcards that occur more than once in the left hand side this may cause a rather slow execution as the number of possible combinations for the wildcard assignments is the factorial of the number of wildcards that are involved.

One of the great applications of Levi-Civita tensors concerns numerical stability problems. Gram determinants are notorious for their numerical instabilities (a Gram determinant is a contraction between two Levi-Civita tensors that have all their non common indices contracted with vectors). Traditionally one contracts the tensors to obtain a number of terms that contain only regular scalar products between the vectors. Such expressions can cause enormous problems during numerical evaluation. In such a

case it is far superior to keep the tensors in and try to compute the product of the two tensors in a different way.

The if statement

FORM is equipped with a number of decision taking instructions. Some operate on the preprocessor level. One of them is the preprocessor **if instruction**. This instruction plays a role in the preparation of the input for the compiler. Its working and syntax are explained in the chapter on the preprocessor. Often one needs to take some decisions during run time. This decision may be based on the particular contents of a term. One of the mechanisms for this is the **if statement**. With this statement it is possible to execute a number of statements selectively, depending on the particular shape of a term. The if statement works a little bit like a hybrid of C and Fortran: there is an if statement, an else statement and an endif statement. Each of these statements must be terminated by a semicolon, as their translation is done by the compiler.

The things that put the if statement in FORM apart from the corresponding statements in C and Fortran are the elements of the condition by which it is decided whether the following statements should be executed. These objects can be :

A number or a fraction
: Any number or fraction that stays within the limitations that govern such quantities are allowed.

coeff[icient]
: This word indicates the coefficient of the current term.

match(pattern)
: The value of the object is the number of times that the pattern can be taken out from the current term. The pattern is

given in the same format as the left hand side of an id statement. This means that it could contain keywords like 'once', 'only', 'many', 'select', etc. The pattern may also contain wildcards. The interpretation of the patterns and the keywords is identical to the interpretation in the id statement, except for the absence of a right hand side.

count(count values)
This gives the count value of the term as obtained when using the defined count values. The syntax of these values is identical to those in the count statement, except for the absence of the first number in the count statement that determines for which count value the term should be discarded.

Objects can be combined via a number of logical operations:

= or ==
The numerical values of the two objects are compared. The answer is TRUE if they are equal.

!=
The numerical values of the two objects are compared. The answer is TRUE if they are not equal.

>
The numerical values of the two objects are compared. The answer is TRUE if the first object is greater than the second.

>=
The numerical values of the two objects are compared. The answer is TRUE if the first object is greater than or equal to the second.

<

The numerical values of the two objects are compared. The answer is TRUE if the first object is less than the second.

<=

The numerical values of the two objects are compared. The answer is TRUE if the first object is less than or equal to the second.

&&

The logical value of the two objects are compared. The answer is TRUE if both objects are TRUE.

||

The logical value of the two objects are compared. The answer is TRUE if one of the objects is TRUE.

Whenever the answer is not TRUE it is FALSE (sic!).

The comparison of two objects is done via the construction <object1> <logical operator> <object2>. An object has a numerical value and a logical value. The numerical value is for instance the value returned by match, count or coefficient. If a numerical value has to be converted to a logical value it is converted to FALSE when the numerical value is zero. It is converted to TRUE when the numerical value is not equal to zero. When a logical value has to be converted to a numerical value FALSE is translated into zero and TRUE is translated into one.

```
if ( coefficient == 10/3 );
   id x = (y+z);
endif;
```

It is possible to chain a number of objects and logical operations without having to use parentheses:

```
if ( match(only,a^2) && match(f?(a?))
                     && match(e_(mu,nu)) );
```

The rule is here that each oject is evaluated and then combined with the total result of everything to the left of it. This may cause some problems:

```
if ( match(a^2*b) == 3 && match(a*b^2) == 4 );
```

This statement will probably be interpreted in a way that is different from what the user wants. First the return value of `match(a^2*b)` is compared with 3. Let us assume that the result is TRUE. Then the return value of `match(a*b^2)` is converted to logical and the logical and operation is taken with the previous result. This gives either in our example the resulting value TRUE. Finally this value TRUE is converted to one and compared to 4. The final answer is always FALSE! The condition can be written properly with the use of parentheses:

```
if ( ( match(a^2*b) == 3 )
  && ( match(a*b^2) == 4 ) );
```

It should be noted that if the first object of a logical 'and' operation gives the value FALSE there is no need to evaluate the second object. This means that in the above example the second match will never be attempted if the first match fails to return the value 3. A corresponding remark can be made about the logical 'if' operation. If the first object returns the value TRUE the second object won't be evaluated.

It is possible to use an if statement without a logical operator. This is done when the condition is rather simple:

example a:

```
s  x, a, b, c;
f  f;
```

```
if ( match(f?(x?)) );
id a*b^2 = 2*c;
endif;
```

In the above example y is replaced by 2*z in all terms that contain at least one function with a single argument that is a symbol. A similar result could be obtained with the statements:

example b:

```
s  x, a, b, c;
f  f;

repeat;
   id f?(x?)*a*b^2 = 2*c*f(x);
endrepeat;
```

With large expressions the user would find out rather soon that the method of example a is much faster. For high powers of a and b it could even be that the solution with the repeat statement can not be used unless the size of the execution heap (see the chapter on the setup parameters) is extended. There is a third solution that avoids the problems of example b but still example a is simpler, faster and easier to understand.

example c:

```
s  x, a, b, c, dummy, dumpow;
f  f;

id a*b^2 = dummy;
id f?(x?)*dummy^dumpow? = f(x)*(2*c)^dumpow;
id dummy = a*b^2;
```

It is harder to find ways around the if statement when the else statement is also used:

```
s  x, a, b, c;
f  f;

if ( ( match(f?(x?)) );
   id a*b^2 = 2*c;
else;
   id a^2*b = 2*c;
endif;
```

It is left to the reader to try to use method b to remove the if statement in the last example.

It is allowed to nest if statements. It is also allowed to place labels inside the 'range' of an if statement. This range consists of the statements between an if and its matching endif. It is not allowed to to put end-of-module instructions inside the range of an if statement.

Limitations: The nesting of if statements is restricted to 10 levels. So is the nesting of parentheses that are used for the grouping inside the conditional part of the if statement. This grouping does not include the parentheses that are used inside match or count operations.

Labels and loops

Like any decent computer language FORM is equipped with the infamous goto statement. For a goto to work properly there must of course also be a label statement. The label statement consists of the word 'label', a blank space, tab or a comma and then a number in the range of 0 to 20. The whole is followed by the ubiquitous semicolon. The syntax of the goto statement is rather similar: The word goto is followed by a blank space, tab or comma after which there is the number of a label that is defined in the same module and then the semicolon. It is not allowed to use the goto to jump to another module. The labels exist only during the execution of the module in which they are defined. This means that the number of a label can be reused again in other modules.

The main uses of labels in FORM are for the construction of loops and to simplify difficult constructions in which otherwise a particular messy piece of code would have to occur more than once. The user can undoubtly think of more applications.

Loops are usually constructed in combination with the if statement. So is the imitation of a while statement given by:

```
Label 0;
if ( condition );
id,......
id,......
etc.

goto 0;
endif;
```

It is not possible to construct do loops that act on the level of

individual terms because we have only preprocessor variables and regular variables. It is however possible so simulate some of these loops rather well:

```
Label 0:
if ( coefficient > 1 );
multiply a/2;


goto 0;
endif;
id a = 2;
```

The above is a loop that will be executed a number of times depending on the size of the coefficient of the term.

It is not allowed to use a goto statement to jump into the range of a repeat/endrepeat statement. It is allowed to jump into the range of an if statement though.

If a loop is executed many times it may place a heavy burden on the stacks and buffers of FORM and it is not unthinkable that an error message will result. In that case one should consult the chapter on the setup file.

Multiply

At times one would like to multiply terms by some expression. Technically such a command isn't needed because on could always multiply the expression by some symbol at the time that it is defined and then replace the symbol by a more complicated object when the time is there, but this is counter intuitive. The multiply statement allows the user to multiply the current term by an expression:

```
multiply,<any expression>;
```

There are two subkeys: left and right.

```
multiply,left,<any expression>;
```

will put the expression at the left of each term, and

```
multiply,right,<any expression>;
```

will put the expression at the right of each term. When the subkey is not present FORM assumes that there are no relevant noncommuting objects and chooses the multiplication that is fastest. This may depend on the installation or the version of FORM. Example:

```
if ( coefficient < 0 );
   multiply, -1;
endif;
```

The above causes all terms to have a positive coefficient.

Gamma Matrices

For its use in high energy physics FORM is equiped with a built-in class of functions. These are the gamma functions of the Dirac algebra which are generically denoted by g_. The gamma matrices fulfill the relations:

```
{g_(j1,mu),g_(j1,nu)} = 2 * d_(mu,nu)
[g_(j1,mu),g_(j2,nu)] = 0    j1 not equal to j2.
```

The first argument is a so called spin line index. When gamma matrices have the same spin line they belong to the same Clifford algebra and commute with the matrices of other Clifford agebra's. The indices mu and nu are over space-time and are therefore usually running from 1 to 4 (or from 0 to 3 in B&D metric). The totally antisymmetric product e_(m1,m2,...,mn)g_(j,m1)... × g_(j,mn)/n! is defined to be gamma5 or g5_(j). The notation 5 finds its roots in 4 dimensional space-time. The unitmatrix is denoted by gi_(j). In four dimensions a basis of the Clifford algebra can be given by:

```
gi_(j)
g_(j,mu)
[g_(j,mu),g_(j,nu)]/2
g5_(j)*g_(j,mu)
g5_(j)
```

In a different number of dimensions this basis is correspondingly different. We introduce the following notation for convenience:

```
g6_(j) = gi(j) + g5_(j)     (from schoonschip)
g7_(j) = gi(j) - g5_(j)
g_(j,mu,nu) = g_(j,mu)*g_(j,nu)  (from reduce)
g_(j,mu,nu,.....,ro,si) =
                  g_(j,mu,nu,.....,ro)*g_(j,si)
g_(j,5_) = g5_(j)
g_(j,6_) = g6_(j)
g_(j,7_) = g7_(j)
```

The common operation for gamma matrices is obtaining the trace of a string of gamma matrices. This is done with the statement:

trace4, j
: Take the trace in 4 dimensions of the combination of all gamma matrices with spin line j in the current term. Any noncommuting objects that may be between some of these matrices are ignored. It is the users responsibility to issue this statement only after all functions of the relevant matrices are resolved. The four refers to special tricks that can be applied in four dimensions. This allows for relatively compact expressions.

tracen, j
: Take the trace in an unspecified number of dimensions. This number of dimensions is considered to be even. The traces are evaluated by only using the anticommutation properties of the matrices. As the number of dimensions is not specified the occurrence of a g5_(j) is a fatal error. In general the expressions that are generated this way are longer than the four dimensional expressions.

It is possible to alter the value of the trace of the unitmatrix. Its default value is 4, but by using the statement

`unittrace value`

it can be altered. Value may be any positive short number (see theinstallation guide for its ranges) or a single symbol with the exception of the symbol i_.

Algorithms:

FORM has been equipped with several built in rules to keep the number of generated terms to a minimum during the evaluation of a trace. These rules are:

rule 0

Strings with an odd number of matrices (not considering gamma5) have a trace that is zero when using trace4 or tracen.

rule 1

A string of gamma matrices is first scanned for adjacent matrices that have the same contractable index, or that are contracted with the same vector. If such a pair is found the relations

```
g_(1,mu,mu) = gi_(1)*d_(mu,mu)
g_(1,p1,p1) = gi_(1)*p1.p1
```

are applied.

rule 2

Next is a scan for a pair of the same contractable indices that has an odd number of other matrices in between. This is done only for 4 dimensions (trace4) and the dimension of the indices must be 4. If found the Chisholm identity is applied:

```
g_(1,mu,m1,m2,...mn,mu) =
          -2*g_(1,mn,...,m2,m1)
```

rule 3

Then (again only for trace4) there is a search for a pair of matrices with the same 4 dimensional index and an even number of matrices in between. If found one of the following variations of the Chisholm identity is applied:

```
g_(1,mu,m1,m2,mu) = 4*gi_(1)*d_(m1,m2)
g_(1,mu,m1,m2,...mj,mn,mu) =
                 2*g_(1,mn,m1,m2,...,mj)
                +2*g_(1,mj,...,m2,m1,mn)
```

rule 4

Then there is a scan for pairs of matrices that have the same index or that are contracted with the same vector. If found the identity:

```
g_(1,mu,m1,m2,...,mj,mn,mu) =
           2*d_(mn,mn)*g_(1,mu,m1,m2,...,mj)
          -2*d_(mu,mj)*g_(1,mu,m1,m2,...,mn)
           ....
        +/-2*d_(mu,m1)*g_(1,mu,m2,...,mj,mn)
        -/+2*d_(mu,mu)*g_(1,m1,m2,...,mj,mn)
```

is used to 'anticommute' these identical object till they become adjacent and can be eliminated via the application of rule 1. After this all gamma matrices that are left have a different index or are contracted with different vectors. These are treated using:

rule 5

Traces in 4 dimensions for which all gamma matrices have a different index, or are contracted with a different fourvector are evaluated using the reduction formula

```
g_(1,mu,nu,ro) =
           g_(1,5_,si)*e_(mu,nu,ro,si)
          +d_(mu,nu)*g_(1,ro)
          -d_(mu,ro)*g_(1,nu)
          +d_(nu,ro)*g_(1,mu)
```

For tracen the generating algorithm is based on the generation of all possible pairs of indices/vectors that occur in the gamma matrices in combination with their proper sign. When the dimension is not specified there is no shorter expression.

Remarks:

When an index is declared to have dimension n and the command trace4 is used the special 4 dimensional rules 2 and 3 are not applied to this index. The application of rule 1 or 4 will then give the correct results. The result will nevertheless be wrong due to rule 5 when there are at least 10 gamma matrices left after the application of the first 4 rules, as the two algorithms in rule 5 give a difference only when there are at least 10 gamma matrices. The result is unpredictable when both indices in four dimensions and indices in n dimensions occur in the same string of gamma matrices. Therefore one should be very careful when using the four dimensional trace under the condition that the results need to be correct in n dimensions. This is sometimes needed when a gamma5 is involved, as the tracen instruction will not allow the presence of a gamma5.

Modulus

Sometimes it is necessary to use coefficients that belong to a finite field. This can be done rather conveniently in FORM. The statement

```
Modulus number;
```

specifies that all arithmatic should be modulus the number 'number'. Of this command only the m is mandatory, the other characters of the word modulus are optional.

When the number is less than the maximal power that is allowed for symbols (see for instance the section on symbols) the powers of symbols and dotproducts are reduced via the relation

```
x ^ number = x;
```

All powers are then positive or zero. The situation with function arguments is somewhat more complicated. They are taken modulus 'number', even though that may be incorrect if one of the arguments represents a power. It is the responsibility of the user to avoid these problems.

The modulus arithmatic extends itself to fractions. These are also reduced to integers, as the inverse of a number can be defined properly unless 'number' is not a prime number. If that is the case a division by zero may result.

When the number that is specified in the modulus statement is larger than the maximum power that is allowed there will be no reduction of powers. The user should work his way around this restriction by the use of special symbols. This is price to be paid for the speed of FORM.

The argument in a modulus statement must be positive or zero. If the argument is 0 or 1 the statement would be meaningless. It is then interpreted as an indication that any existing modulus arithmatic must be switched off and ordinary fractional arithmatic should be restored.

Sometimes the reduction of powers in modular arithmatic can be a nuisance. In that case this reduction can be turned off by specifying a negative value in the modulus statement. So

```
Modulus,-17;
```

will set all arithmatic of coefficients to modulus 17, but exponents will be unaffected. Note that the comma is necessary here.

When the value in the modulus statement is a prime number there exists another feature that is handy at times. It is possible to find a number x so that when arithmatic is modulus p all numbers greater than zero and less than p can be written as x^n for some n. Such a number x is called a generator. The user can select the option to print all coefficients as a power of a generator with the statement

```
Modulus p:x;
```

FORM will then construct a table in which each number less than p is expressed as a power of the generator. This will fail when the generator turns out to be not a real generator or when there is not enough memory for such a table. Therefore this option is only useful when p is not very large.

Procedures

Like any decent language FORM can use prepared input that depends on a number of parameters. Such input is often referred to as subroutines or procedures. The procedures in FORM are managed by the preprocessor. This means that, unlike in the computational languages, a procedure is not necessarily called just to produce a number. It is called to act upon the current environment in a way that depends on the contents of the procedure and the specific 'value' of the parameters. A procedure can exist of just a collection of id statements or it can define new expressions etc.

There are two ways to define procedures: The first one is at the beginning of a program. If the preprocessor encounters a procedure instruction the contents of the procedure are copied into a procedures buffer. The end of the procedure is indicated with an 'endprocedure' instruction. This way any number of procedures can be defined. The requirement is that a procedure must be defined before it is used and the occurrence of the definition of a procedure inside the range of a loop may cause a fatal error.

The second method is to define the procedure in a prepared file. This file must have the name of the procedure to which the extension '.prc' is added. If the filesystem does not support extensions to file names the period is omitted and the characters 'prc' are added. So the procedure 'solve' should be stored in the file 'solve.prc' (or 'solveprc' on 'those other' systems).

The first line of a procedure must be the procedure declaration and it should also contain the parameter field definition. This line must be a single line, so if the user likes to extend it over more than a single inputline the linefeed should be escaped with a backslash.

The syntax of the line is:

```
#procedure name(arg1,arg2,...,argn)
```

The arguments must have names that consist of regular alphanumerical characters. They are separated by comma's. Also the name of the procedure must consist of alphanumerical characters only. The procedure declaration must be in the first line of the file when it is defined in a file. Note that FORM is very picky about the name of the procedure. It must be exactly equal to the name that was used to obtain the name of the file (when defined in a file), even though this implies some redundancy. The end of the procedure is indicated with the 'endprocedure' instruction which has the very simple syntax

```
#endprocedure
```

This instruction is used only when the procedure is defined at the beginning of a program. When it is encountered in a file it is skipped.

A procedure is used with the 'call' instruction. This instruction has the syntax:

```
#call name {parameter1|parameter2|...|parameter}
```

Here the parameterfield obeys the same rules as the parameter field in the listed do loops, ie it is enclosed in curly brackets (`{` and `}`) and the parameters are separated by a vertical line (`|`). This notation allows the use of any regular characters as parameters, including strings that contain comma's or linefeeds. If a parameter contains curly brackets, these brackets should be paired properly inside each parameter separately.

The parameters inside the procedures are local preprocessor variables. This means that inside the procedure the local definition takes precedence over a potentially already existing definition.

In other words: if the variable i is used in a do loop and a procedure, which uses a variable i, is called inside this loop there will be no conflict. Inside the procedure the local definition is used, while after the completion of the procedure the original definition is active again. Note also that the parameters are preprocessor variables so they should be enclosed in quotes when they are used.

For some applications of procedures one should consult the chapter with the examples.

Saving and Loading

Often it is necessary to save the results of one program so that it may be used in another program. This process is called saving. In FORM expressions can be saved if they are already stored expressions. The save statement is given either with a list of expressions that should be saved or without such a list in which case all stored expressions are saved:

```
save filename;
save filename expressions;
```

The name of the file should be acceptable to the filesystem of the computer on which FORM is running. The expressions should all be valid expressions that were properly stored.

To use the saved expressions in another program one should apply the load statement. Its syntax is similar to the syntax of the save command:

```
load filename;
load filename expressions;
```

In the first case all expressions in the given file are loaded, while in the second case only the expressions that are asked for are loaded. If an expression is not found or when loading would cause an expression to occur twice a fatal error is generated. A fatal error is also generated when a loaded expression is being used and its namelists are incompatible with the current namelists. For each expression that is loaded FORM will print a line with the name of the expression. This allows the user to check the contents of a file. There is no restruiction on the number of save and load

statements in a program. It is suggested that the extension of a file with saved expressions is .sav on systems that allow file names with extentions.

The internal storage file in which the stored expressions are kept is written in the same format as the 'save file'. This means that if a program is aborted prematurely all information in the storage file can be recovered, unless an error occurred during the updating of the storage file. The storage file is removed after successful completion of a program. When it is not deleted it can be found either in the current directory or in the directory designated to contain the temporary files. Its name will end either in .str on systems that allow extentions to file names or in str when such extentions are not allowed.

The save and load instructions are executed by the compiler, so when such an instruction is found between a number of id statements it is still executed only once. The compiler output will show no trace of the presence of a save or load instruction.

The setup parameters

When FORM is started it has a number of settings built in that were determined during its installation. If the user would like to alter these settings he can specify their desired values in a setup file. There are two ways in which FORM can find a setup file. The first way is by having a file named 'form.set' in the current directory. If such a file is present FORM will open it and interpret its contents as setup parameters. If this file is not present one may specify a setup file via the -s option in the command tail. This option must preceede the name of the input file. After the -s follow one or more blanks or tabs and then the full name of the setup file. FORM will try to read startup parameters from this file. If a file 'form.set' is present FORM will ignore the -s option and its corresponding file name. This order of interpretation allows the user to define an alias with a standard setup file which can be overruled by a local setup file. If neither of the above methods is used FORM will use a built in set of parameters. Their values may depend on the installation and are given in the installation guide.

The following is a list of parameters that can be set. The syntax is rather simple: The full word must be specified (case insensitive), followed by one or more blanks or tabs and the desired number. Anything after the number is considered to be commentary. Also lines that don't start with an alphabetic character are seen as commentary. The order of description is the order in which the parameters are used during the memory allocation phase.

MaxTermSize

This is the maximum size that an individual term may oc-

cupy. This size doesn't affect any allocations. One should realize however that the larger this size is the heavier the demand can be on the workspace. Consequently the evaluationtree cannot be very deep when WorkSpace / TermSize is not very big. MaxTermSize controls mainly when FORM starts complaining about terms that are too complicated. The size is in a number of short integers.

NwriteStatistics

When this word is mentioned the default setting for the statistics is that no run time statistics will be shown. Ordinarily they will be shown.

LineLength

The number that follows determines a cutoff on the number of characters that will be printed on one line. This rule is not adhered to rigidly as some types of output may take a few extra characters. This LineLength should not be outside the range 39 to 255 or it will be corrected to a number inside this range. This parameter looks much like the Format parameter that can be set during run time, except for that the LineLength cannot take symbolic values like 'fortran'.

ConstIndex

This is just the number of indices that are considered to be constant indices like in fixed vector components. The size of this parameter is not coupled to any array space, but ConstIndex+4*Vectors+4*Indices) must be a number that fits inside a short integer in FORM (usually two bytes).

TempDir

This variable should contain the name of a directory that is the directory in which FORM should make its temporary

files. If the -T option is used when FORM is started the TempDir variable in the setup file is ignored. FORM can create 5 different temporary files. They have the extensions .str, .sc1, .sc2, .sor and .sga. The first file contains the global expressions after they have been stored away by a .store instruction. The internal format is identical to the format of the files that are written by the 'save' statement. When FORM aborts execution and there is a possibility that the .str file contains relevant information it is not removed. The other files are removed at the end of execution. The .str file can then be read by a 'load' statement.

PreBuffer
The size of the buffer that the preprocessor uses for storing preprocessor variables.

PreVariables
The number of variables that the preprocessor can have in its buffers simultaneously.

Symbols
The maximum number of symbols that can be defined at any given moment. Note that a complex symbol takes two entries: one for the symbol itself and one for its complex conjugate.

Indices
The maximum number of indices that can be defined at any time.

Vectors
The maximum number of vectors that can be defined at any time.

Functions
The maximum number of functions that can be defined at any time. Again the complex functions use two spots in the tables.

Sets
The maximum number of sets that can be defined at any time.

SetStore
The maximum number of elements in all sets together.

Expressions
The maximum number of expressions that can be defined at any time.

MaxNames
The compiler part of FORM keeps a list of pointers to all existing names of variables. This enables the compiler to find names rather quickly. The size of this list is restricted by the variable MaxNames. If the user needs more than this number of names he should increment the value of MaxNames.

NameBuffer
The maximum storage space for all names in the name tables. Each name occupies a number of bytes that is equal to the number of characters in the name plus one termination character and if this sum is odd there is an extra character to make the total length even.

MaxWildcards
The maximum number of wildcards that can be active in a single matching of a pattern.

InputSize

The size of the raw input buffer. This buffer is used to collect an entire statement before it is sent to the compiler. This size is therefore a limitation on the size of a single instruction. The size that is limited is the size that the statement obtains after the preprocessor variables have been expanded. This buffer is also used during the execution of a program. It serves for temporary storage of some quantities during wildcard expansions. This involves rarely more than a few hundred bytes.

CompilerOutput

The size of the buffer that contains the output of the compiler. This buffer contains the output that concerns an entire module. This compiler output is used directly by the algebra processor. The algebra processor sets up a single tree for each term in the input so it would be very wasteful to have the compiler output residing on disk. Therefore the compiler output must be in memory. If the allocated memory is not sufficient and more memory cannot be obtained the user should try to simplify his modules by breaking them up.

MaxStatements

The maximum number of executable statements that is allowed in each module.

MaxLevels

The maximum number of effective statements that is allowed in a module. The number of effective statements is obtained by taking the number of executable statements, adding to that the number of pairs of parentheses and adding to that the number of nontrivial function arguments.

FunctionLevels
The maximum number of levels that may occur when functions have functions in their arguments.

MaxBracket
The maximum number of words that can be used to store bracket information. If this number is too small the bracket statement will issue an error message.

FilePatches
The maximum number of patches that is allowed on a file during sorting. Currently this number is limited because the file sorting is done in one pass. There is a correllation between this number and the size of the other buffers: Each patch will be buffered with a buffer that must have at least the maximum size of a term. In addition there must be space for rational arithmatic, so each buffer must have a minimum size of (bytes in MaxTermSize) $*$ 2. In most computers this means that FilePatches must be less than (LargeSize+SmallExtension)/(4$*$MaxTermSize). Actually the size of these buffers is fixed by SortIOsize.

LargePatches
The maximum number of patches that is allowed in the large buffer. The large buffer resides in virtual memory and whenever it is full or the number of patches exceeds the maximum it is sorted and the result is written to the intermediate file that is used for sorting.

TermsInSmall
The maximum number of terms that is allowed in the small buffer before it is sorted. The sorted result is either copied to the large buffer or written to the intermediate sort file.

SmallSize

The size of the small buffer in bytes. This buffer is used to place the freshly generated terms in. When it is full or its number of terms exceeds a maximum number (TermsInSmall) or the generation of terms is finished for the current expression the terms in this buffer are sorted and the results are sent to either the large buffer, the intermediate sort file or to the output.

SmallExtension

The size of the small buffer plus its extension. The small buffer needs an extension that is contiguous with it, as during the sort the sum of two terms may take more space than either of the individual terms. This is the price to be paid for arithmatic with rational numbers. Such a term can then be moved to the extension space and the sort continues. If the extensionspace is also full a garbage collection is attempted. This garbage collection uses the disk so it is not very fast. Typical settings are that SmallExtension is about 1.25 times the size of SmallSize. When SmallSize is set SmallExtension is automatically set to such a value. So it is useless to first specify SmallExtension and then SmallSize. A value for SmallExtension that is less than SmallSize results in an error message.

LargeSize

The size of the large or virtual buffer. This buffer is useful on systems with virtual memory that is much larger than the physical memory. Sorting in the small buffer is done by pointers. This method works only well when the whole small buffer resides inside the physical memory. When virtual memory is going to be used a different sorting mechanism is applied. It uses the pages only once in their entirety, but it

has to move terms. This sort merges at most 'LargePatches' patches that are each the result of a sort inside the small buffer. The results are either put in the intermediate sort file or in the output file. Some experimenting may be needed to determine the optimal size of LargeSize and SmallSize. This optimum may also be function of the workload on the machine. The large buffer and the small buffer are assigned to be contiguous. This gives a maximal size for the buffers that can be used when the intermediate sort file is merged. (See also MaxPatches).

SortIOsize

The size of the buffer that is used to write to the intermediate sorting file. This buffer must have a size that is at least as large as the MaxTermSize $*$ (bytes per short integer) and (SortIOsize+MaxTermSize in bytes) $*$ MaxPatches must be less than SmallExtension + LargeSize.

ScratchSize

The size of the input and the output buffers for the regular algebra processing. Terms are read in in chunks this size and are written to the output file via buffers of this size. There are two of these buffers. These buffers must have a size that is at least as large as the MaxTermSize $*$ (bytes per short integer).

NumberStoreCache

See SizeStoreCache below.

SizeStoreCache

When stored expressions are used they are read term by term. This may be rather slow on some computers. Therefore some caching mechanism has been built in. The first

'NumberStoreCache' expressions get a chaching buffer assigned to them. These buffers are 'SizeStoreCache' bytes long. For the purpose of counting expressions each expression contributes by the number of its power, so `F^2` uses two cache buffers. These buffers must have a size that is at least as large as the MaxTermSize $*$ (bytes per short integer).

WorkSpace

The size of the heap that is used by the algebra processor when it is evaluating the substitution tree. It will contain terms, half finished terms and other information. The size of the workspace may be a limitation on the depth of a substitution tree.

Restrictions: The small buffer and the large buffer are used by FORM as a collection of input buffers when sorting gets to the stage that the patches in the intermediate sorting file have to be merged. This can only be done if there is enough room in this buffer for one input buffer for each patch on the file plus some overflow space to avoid having only parts of a term in the buffer. This means that:

```
SmallExtention + LargeSize >=
   LargePatches$*$(SortIOsize +
   MaxTerSize$*$bytes per short integer).
```

The current default settings should be looked up in the installation guide.

The file `setup` that comes with FORM shows the default settings. It is an example of a file that contains all parameters. In practice one only has to specify those parameters that must be different from their default value.

Example file `form.set`:

```
MaxTermSize         1000
```

```
  NwriteStatistics is off
LineLength            79
ConstIndex           128
TempDir
PreBuffer            200
PreVariables          20
Symbols              100
Indices              100
Vectors              100
Functions            100
Sets                  50
SetStore             200
Expressions          100
MaxNames             250
NameBuffer          4096
MaxWildcards          40
InputSize          32768
CompilerOutput     50000
MaxStatements        100
MaxLevels            500
FunctionLevels        10
MaxBracket           200
FilePatches           64
LargePatches         128
TermsInSmall       20000
SmallSize         450000
SmallExtension    600000
LargeSize        2000000
SortIOsize          8192
ScratchSize         8192
NumberStoreCache       5
SizeStoreCache      4096
WorkSpace          50000
```

Bug hunting

However careful one may be, occasionally there are those mysterious error messages that take a while to figure out. Take for instance the following program:

```
    Symbols x,n;
    CFunctions g1,g2
    Functions f1,f2,f,dx;
    Local F = f1(0,x)*f2(0,x);
    multiply dx;
    repeat;
     id,dx*f?(n?,x) = f(n+1,x)+f(n,x)*dx;
    endrepeat;
    id dx = 0;
    print;
    .end
WorkSpace overflow      40000 bytes is not enough.
```

It is also possible that this program doesn't even yield an error message but that a stack overflow occurs first having FORM crash quite unelegantly (there is no check for stack overflow currently). What went wrong here? The program looks very much like one of the programs we made in one of the sections in the tutorial.

One of the ways in which FORM can help us is by printing how it interpreted the name lists. We can force it to do so with the 'write names' statement. This statement turns on a flag that causes FORM to let us have a look at the namelists each time that FORM encounters an end of module instruction. This flag can be turned off again with the 'nwrite names' statement.

```
    Symbols x,n;
    CFunctions g1,g2
    Functions f1,f2,f,dx;
    Local F = f1(0,x)*f2(0,x);
    multiply dx;
    repeat;
     id,dx*f?(n?,x) = f(n+1,x)+f(n,x)*dx;
    endrepeat;
    id dx = 0;
    write names;
    print;
    .end

 Symbols
   i_#i x n
 Functions
   g_#i sum_ sump_
 Commuting Functions
   e_#i fac_ theta_ delta_ g1 g2 Functions f1 f2 f
   dx
 Expressions
   F(local)
 Expressions to be printed
   F
WorkSpace overflow      40000 bytes is not enough.
```

Now what do we see: Our functions have been entered in the namelists as commuting functions, even the ones that we declared as noncommuting functions. In addition there is the mysterious function 'Functions'. Let us have a close look at the function declarations. By now the reader may notice (if he hadn't from the beginning) that there is a semicolon missing in the declaration of the commuting functions. So form read the next line as a

continuation of the earlier line. There is very little that can be done to prevent such a situation. The freedom that the syntax of FORM gives to the user may occasionally result in an error that isn't caught by the compiler. On the other hand the above case is rather rare.

What happens more often is that the user forgets a semicolon, after which the compiler will issue a message about incorrect syntax of a statement that doesn't refer to semicolons at all. Therefore one of the guidelines should be that if the error message seems to be about something that looks correct, one should inspect whether all statements in the neighborhood are terminated properly.

Another common error is the error 'Write error while sorting' after which FORM will terminate the execution. This indicates usually that either the user tried to write beyond he quota (on mainframes) or that his disk is full. Inspection of the statistics should reveal whether such is the case. On UNIX systems the use of the 'TempDir' option could solve this problem by having the intermediate files written to the directory /tmp (or in spool).

List of commands

The following is a list of all statements, instructions, built in functions and options. Note that FORM is case insensitive with respect to these objects. If the keyword can be abbreviated the mandatory part is outside the square brackets, while the optional part of the word is inside the square brackets. Note that the optional part may be fractionally present. D dim dimen or dimension all mean the same.

A
See 'Symbols'.

Al[so]
See 'Idold'.

An[tisymmetrize]
Antisymmetrize the given function according to the specified parameter field information.

B[rackets]
Should be followed by a list of objects that must be taken outside parentheses when the output is printed.

C[functions] or C[ommuting]
Declaration of commuting functions.

Coeff[icient]
Function that returns the value of the coefficient of the current term.

Contract
Contraction of Levi-Civita tensors.

Count
Function for power counting. When used as a statement all terms that don't fulfil the count condition are discarded.

Delete Storage
Deletes the contents of the storage file.

delta_
The delta function with values 0 or 1.

D[imension]
Set the default dimension.

Discard
Discards the current term.

Drop
Skip the given expressions and remove them after the next .sort instruction.

d_
The Kronecker delta.

else
See if-statement.

endif
See if-statement.

Endrepeat
End of a group of statements that should be repeated till no more replacements are made.

e_
The Levi-Civita tensor.

Fi[xindex]
Set the value of d_ for the diagonal elements with number indices.

Fo[rmat]
Set either the number of columns per line or set in Fortran mode

F[unctions] or N[functions]
Declaration of non commuting functions.

G[lobal]
Definition of a global expression.

Go[to] or Go[to]
Go to the label who's number is given after the go to.

g_
The built in gamma matrix.

i_
The imaginary i.

Id[entify] or Id[new]
Replacement statement. Enters a new level which means that the matching of the lhs will be made after the preivous statement has been completed.

Ido[ld] or Al[so]
Replacement statement. The matching of the lhs takes place before the rhs of the previous Id or Idold statement has been inserted.

If
The if statement that reacts to the contents of terms.

I[ndex] or I[ndices]
Declaration of indices.

La[bel]
Put a label before the next statement.

L[ocal]
Definition of a local expression.

Loa[d]
Load expressions from an external file into the file with stored expressions.

Many
Option in id-statement. Take pattern once at a time but then as many times as possible.

Match
Function that returns the number of times that the given pattern fits in the current term.

M[odulus]
Change to modulus arithmatic.

Multi
Option in id-statement. Take pattern as many times as possible.

Mu[ltiply]
Multiply each term by a symbol.

N[functions]
See Functions.

Np[rint]
Don't print the specified expressions.

Nw[rite]
Don't write names or statistics.

Once
Option in id-statement. Take pattern only once.

Only
Option in id-statement. Take pattern only if exact match.

P[rint]
Print the specified expressions.

Ratio
Partional fractioning with respect to given objects.

Repeat
Repeat the group of statements between the repeat and the matching endrepeat statement till there are no more replacements.

Sa[ve]
Save expressions from the file with stored expressions to an external file.

Select
Option in id-statement. Take pattern only if no elements of the additionally mentioned sets are left.

Skip
The given expressions will not be the object of operations in this module.

Sum
Declare that an index is summed over.

sum_
The built in sum function.

sump_
The built in sum function with special incrementation rules.

S[ymbols] or A
Declaration of symbols.

Symm[etrize]
Symmetrize the given function according to the specified parameter field information.

theta_
The theta function with values 0 or 1.

Tracen
Take trace of gamma matrices using n-dimensional algorithms.

Trace4
Take trace of gamma matrices using 4-dimensional algorithms.

U[nittrace]
Set the trace of the unit matrix gi_.

V[ectors]
Declaration of vectors.

W[rite]
Write names or statistics.

Se[t]
Declaration and definition of a set.

.clear
Clears all the contents of FORMs buffers with the exception of the input and preprocessor buffers.

.end
End of module and end of program.

.global
End of module. No evaluations are made. Make the declarations and definitions in this module global.

.sort
End of module. Evaluate all expressions.

.store
End of module. Evaluate all expressions. Store the global expresssions and remove all local expressions.

#call
Call of a procedure.

#define
Define a preprocessor variable.

#do
Preprocessor do-loop.

#else
Preprocessor else instruction.

#enddo
End of preprocessor do-loop.

#endif
Preprocessor endif instruction.

#endprocedure
End of the definition of a procedure.

#if
Preprocessor if instruction.

#include
Insert the contents of the mentioned file into the input.

#procedure
Definition of a procedure.

#undefine
Undefine a preprocessor variable.

#+
Resume listing the input.

#-
Stop listing the input.

*
Line is commentary.

Examples

27.1 Polynomial substitutions

Usually polynomials in commuting objects are substituted using binomial coefficients when powers of the polynomials are involved. A simple example is:

```
s     a,b;
L     F=(a+b)^4;
print;
.end

   F =
      a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4
```

In this case FORM generated only the five terms in the output, rather than 16 terms. When there are noncommuting objects (functions, subexpressions containing noncommuting functions, or expressions) involved the brute force method is used, rather than the binomial expansion.

In the substitution of one Taylor expansion inside another expansion even the use of binomial coefficients is not sufficient. An expression containing m terms raised to the n-th power will still generate (n+m-1)!/(n! (m-1)!) terms before FORM can collect and sort them. When n and m are of the order of 20 such a method is clearly impractical.

The practical method will be illustrated in the following example. We are going to take the expansion of ln(1+x) to 40 terms deep and substitute in it the expansion for exp(z)-1. The result

should of course be z after lengthy computations. This trivial result allows us to check that the algorithm is correct and also that the arithmatic routines of FORM are functioning properly.

The expansions we need are:

$$\ln(1+x) = \sum_{i=1}^{n} (-1)^{i+1} x^i / i$$

and:

$$e^z - 1 = \sum_{i=1}^{n} z^n / n!$$

This last expansion has to be modified, because when n is rather large the results can be catastrophic. We will use the 'telescope' formula:

$$e^z - 1 = z(1 + \frac{z}{2}(1 + \frac{z}{3}(1 + \frac{z}{4}(1 \cdots))))$$

This formula is applied by the successive substitutions:

```
id   x = y*z;
id   y = 1 + y*z/2;
id   y = 1 + y*z/3;
id   y = 1 + y*z/4;
       .
       .
       .
id   y = 1 + y*z/n;
id   y = 1;
```

The use of the preprocessor do loops makes it rather convenient to program. Note also the use of the preprocessor variable n to indicate the depth of the expansion and the cutoff on the power of z.

```
#define n "40"
s    x,y,z(:'n'),i;
L    F = sum_(i,1,'n',-(-x)^i/i);
id   x = y*z;
#do i = 2,'n'
id   y = 1 + y*z/'i';
#enddo
id   y = 1;
print;
.end
```

This program gives the answer z, but it takes still quite some time to do so, as it still generates 215307 terms, disregarding the terms that were discarded prematurely because they had a power of z that was larger than 40. The program becomes nearly 40 times faster when this build up is suppressed by sorting the terms after each substitution. This is done by adding a .sort statement inside the last loop:

```
#do i = 2,'n'
id   y = 1 + y*z/'i';
.sort
#enddo
```

27.2 Solving equations

Many algebra programs contain built in commands to solve sets of equations. The drawback of such commands is that they choose often an inefficient algorithm because the algorithm must be general. FORM doesn't have built in commands to solve equations but it provides the tools to build ones own procedures to solve a particular class of equations. In this example we will solve a set of three equations in three variables. This is best done in a procedure, so that we may use it whenever we run again into the same problem.

27.2.1 Method 1

The first method is rather similar to the way one uses when solving the equations by hand. We eliminate variables one by one to obtain equations with fewer variables. This is done in the following procedure:

```
#procedure solve3(F1,F2,F3,x1,x2,x3,G1,G2,G3)
*
*  Solves 3 linear equations in 3 variables.
*  F1, F2, F3 are the three equations.
*  x1, x2, x3 are the variables.
*  G1, G2, G3 are the equations with the results.
*
*  Make first two equations in two variables:
*
b  'x3';
nprint;
.sort
L  Solve3Aux1 = 'F1'*'F2'['x3'] - 'F2'*'F1'['x3'];
L  Solve3Aux2 = 'F1'*'F3'['x3'] - 'F3'*'F1'['x3'];
*
```

```
*  Now eliminate first x2 to obtain x1:
*
b  'x2';
nprint;
.sort
L  'G1' = Solve3Aux1*Solve3Aux2['x2']
        - Solve3Aux2*Solve3Aux1['x2'];
*
*  Next eliminate first x1 to obtain x2:
*
b  'x1';
nprint;
.sort
skip 'F3';
drop Solve3Aux1,Solve3Aux2;
L  'G2' = Solve3Aux1*Solve3Aux2['x1']
        - Solve3Aux2*Solve3Aux1['x1'];
*
*  Now something similar to obtain x3
b  'x1';
nprint;
.sort
skip 'G1','G2','F1','F2','F3';
L  Solve3Aux1 = 'F1'*'F2'['x1'] - 'F2'*'F1'['x1'];
L  Solve3Aux2 = 'F1'*'F3'['x1'] - 'F3'*'F1'['x1'];
b  'x2';
nprint;
.sort
skip 'F1','F2','F3';
drop Solve3Aux1,Solve3Aux2;
L  'G3' = Solve3Aux1*Solve3Aux2['x2']
        - Solve3Aux2*Solve3Aux1['x2'];
b  'x1','x2','x3';
```

```
print;
.sort
nprint;
```

This procedure should be stored in the file 'solve3.prc' (or 'solve3prc' when extentions with a period are not allowed) and can be used from then on. It is used in the following program:

```
s   a1,b1,c1,d1,a2,b2,c2,d2,a3,b3,c3,d3,x,y,z;
nwrite statistics;
L   V1 = a1*x+b1*y+c1*z+d1;
L   V2 = a2*x+b2*y+c2*z+d2;
L   V3 = a3*x+b3*y+c3*z+d3;
#call solve3{V1|V2|V3|x|y|z|H1|H2|H3}
.end
```

and the results are printed. Of course the results are not as nice as could be when printing 'x =' etc., but on the other hand we are not harmed by sets of equations that don't have a solution! Nevertheless running the program shows a considerable weakness: The resulting formulae have each a factor by which they can be divided. This means for instance that if c1 is zero the first two equations (for x and y) become meaningless. The better way is to use the formal approach that follows below.

27.2.2 Method 2

A set of linear equations can always be written in terms of matrices using the formula

$$A.x = b$$

in which x and b are vectors and A is a matrix. The solution of these equations is obtained by multiplying with the inverse of A:

$$x = A^{-1}.b$$

The inverse of A is given by its minors divided by the determinant, so the following program will do the job:

```
#procedure solve3a(F1,F2,F3,x1,x2,x3,H1,H2,H3)
nprint;
b  'x1','x2','x3';
.sort
skip 'F1','F2','F3';
s  Solve3DetAux;
L  'H1' = 'x1' * Solve3DetAux
        + 'F1'['x2']*'F2'['x3']*'F3'[1]
        - 'F1'['x2']*'F2'[1]*'F3'['x3']
        - 'F1'['x3']*'F2'['x2']*'F3'[1]
        + 'F1'['x3']*'F2'[1]*'F3'['x2']
        + 'F1'[1]*'F2'['x2']*'F3'['x3']
        - 'F1'[1]*'F2'['x3']*'F3'['x2'];
L  'H2' = 'x2' * Solve3DetAux
        - 'F1'['x1']*'F2'['x3']*'F3'[1]
        + 'F1'['x1']*'F2'[1]*'F3'['x3']
        + 'F1'['x3']*'F2'['x1']*'F3'[1]
        - 'F1'['x3']*'F2'[1]*'F3'['x1']
        - 'F1'[1]*'F2'['x1']*'F3'['x3']
        + 'F1'[1]*'F2'['x3']*'F3'['x1'];
L  'H3' = 'x3' * Solve3DetAux
        + 'F1'['x1']*'F2'['x2']*'F3'[1]
        - 'F1'['x1']*'F2'[1]*'F3'['x2']
        - 'F1'['x2']*'F2'['x1']*'F3'[1]
        + 'F1'['x2']*'F2'[1]*'F3'['x1']
        + 'F1'[1]*'F2'['x1']*'F3'['x2']
        - 'F1'[1]*'F2'['x2']*'F3'['x1'];
id Solve3DetAux =
          'F1'['x1']*'F2'['x2']*'F3'['x3']
        - 'F1'['x1']*'F2'['x3']*'F3'['x2']
```

```
        - 'F1'['x2']*'F2'['x1']*'F3'['x3']
        + 'F1'['x2']*'F2'['x3']*'F3'['x1']
        + 'F1'['x3']*'F2'['x1']*'F3'['x2']
        - 'F1'['x3']*'F2'['x2']*'F3'['x1'];
b  'x1','x2','x3';
print;
.sort
nprint;
```

Of course typing in such a procedure can be quite a nuisance, unless one types it in the 'normal' way and uses a decent editor to replace the shorthand object by the objects with the quotes. There is however still a better way:

27.2.3 Method 3

The above method used the determinant of a three by three matrix. Actually the solution of our problem can be expressed in terms of three regular determinants (multiplied by x1, x2 and x3) and the three determinants that are obtained by replacing one of the columns of the matrix A by the column vector b. So if we make one procedure with a three by three determinant we can call it six times to solve our problem. The determinant is given by:

```
#procedure det3(F1,F2,F3,x1,x2,x3)
  (
  + 'F1'['x1']*'F2'['x2']*'F3'['x3']
  - 'F1'['x1']*'F2'['x3']*'F3'['x2']
  - 'F1'['x2']*'F2'['x1']*'F3'['x3']
  + 'F1'['x2']*'F2'['x3']*'F3'['x1']
  + 'F1'['x3']*'F2'['x1']*'F3'['x2']
  - 'F1'['x3']*'F2'['x2']*'F3'['x1'];
  )
```

```
#endprocedure
```

The new routine to solve the equations becomes:

```
#procedure solve3b(F1,F2,F3,x1,x2,x3,H1,H2,H3)
nprint;
b  'x1','x2','x3';
.sort
skip 'F1','F2','F3';
L  'H1' = 'x1' *
#call det3{'F1'|'F2'|'F3'|'x1'|'x2'|'x3'}
        +
#call det3{'F1'|'F2'|'F3'|1|'x2'|'x3'}
        ;
L  'H2' = 'x2' *
#call det3{'F1'|'F2'|'F3'|'x1'|'x2'|'x3'}
        +
#call det3{'F1'|'F2'|'F3'|'x1'|1|'x3'}
        ;
L  'H3' = 'x3' *
#call det3{'F1'|'F2'|'F3'|'x1'|'x2'|'x3'}
        +
#call det3{'F1'|'F2'|'F3'|'x1'|'x2'|1}
        ;
b  'x1','x2','x3';
print;
.sort
nprint;
#endprocedure
```

Note that the parameter 1 will cause the element `F1[1]` to be generated which is the part of the expression F1 which has nothing outside parentheses. The result of the above program is now:

```
s  a1,b1,c1,d1,a2,b2,c2,d2,a3,b3,c3,d3,x,y,z;
L  V1 = a1*x+b1*y+c1*z+d1;
L  V2 = a2*x+b2*y+c2*z+d2;
L  V3 = a3*x+b3*y+c3*z+d3;
#call solve3b{V1|V2|V3|x|y|z|H1|H2|H3}

   H1 =
       + x * ( a1*b2*c3 - a1*c2*b3 - b1*a2*c3 +
         b1*c2*a3 + c1*a2*b3 - c1*b2*a3 )
       + b1*c2*d3 - b1*d2*c3 - c1*b2*d3 + c1*d2*
         b3 + d1*b2*c3 - d1*c2*b3

   H2 =
       + y * ( a1*b2*c3 - a1*c2*b3 - b1*a2*c3 +
         b1*c2*a3 + c1*a2*b3 - c1*b2*a3 )
       - a1*c2*d3 + a1*d2*c3 + c1*a2*d3 - c1*d2*
         a3 - d1*a2*c3 + d1*c2*a3

   H3 =
       + z * ( a1*b2*c3 - a1*c2*b3 - b1*a2*c3 +
         b1*c2*a3 + c1*a2*b3 - c1*b2*a3 )
       + a1*b2*d3 - a1*d2*b3 - b1*a2*d3 + b1*d2*
         a3 + d1*a2*b3 - d1*b2*a3

  .end
```

It is clear that this method can be extended rather easily to different numbers of equations and unknown quantities. The fact that writing out larger determinants becomes rather tiresome reflects somewhat the inefficiency of solving a large number of equations via this method. Usually there are different tricks when solving such a big system of equations. The fastest ways are then hybrid ways involving inspection and intelligent action of the user

while FORM will do the work. It is also possible to use one of the other ways we have seen to evaluate determinants, but they will only simplify the determinant routine. The speed of the program won't be better as the amount of work for evaluating the determinant has not deminished.

27.3 Determinants of sparce matrices

When one has to take the determinant of a matrix it occurs often that the matrix contains many zeroes. Taking the determinant then by means of a general method (like the contraction of two Levi-Civita tensors) can be very wasteful. The general method is bound to generate the maximum number of terms, even though most of them are thrown away rather quickly. When such a determinant is evaluated by hand (for as far as possible) the zeroes are used to eliminate much of the work. In this example we will evaluate a two Sylvester determinants that may occur when finding the common solution space of two quartic equations (gives an 8 by 8 determinant) and when finding the condition for simultaneous solutions of a quartic and a fifth order equation (which gives a 9 by 9 determinant). Even this last determinant requires no more than two minutes on a small computer!

When doing a determinant by hand a popular strategy is to select the row with the maximum number of zeroes and use this row to construct (the minimum number of) (n-1) by (n-1) minors. This procedure can be addapted again and again until there are only trivial matrices left. We will construct a procedure here that works this way. Because we cannot make procedures with a variable number of arguments the procedure has to be addapted a little bit when we change the size of the matrix.

Let us assume that the rows of the matrix have been ordered such that the row with the least amount of zeroes is the top row, the next row has the second largest number of zeroes etc. If we want to indicate a i by i minor that is left after n-i rows have been treated it is sufficient to give the columns of that minor. So considering an 8 by 8 matrix M the minor `M(2,3,4,6,8)` is one of the minors that is left after three rows have been 'eliminated'. The whole matrix is indicated by `M(1,2,3,4,5,6,7,8)`. The rule for breaking a 5 by 5 determinant down to 4 by 4 determinants is

very simple:

```
id M(m1?,m2?,m3?,m4?,m5?) =
    + matrix(1,m1)*M(m2,m3,m4,m5)
    - matrix(1,m2)*M(m1,m3,m4,m5)
    + matrix(1,m3)*M(m1,m2,m4,m5)
    - matrix(1,m4)*M(m1,m2,m3,m5)
    + matrix(1,m5)*M(m1,m2,m3,m4);
```

The main problem is now to program such a reduction statement in a way that it can be used for any size matrix and for any step in the computation. Below is one way of doing this. It uses the preprocessor rather heavily:

```
#procedure peeldet(ii,M,N,n1,n2,n3,n4,n5,n6,n7,n8)
id  'M'(
#do iii = 1,'ii'
    m'iii'?
#enddo
    ) =
#do iii = 1,'ii'
    + {2*('iii'%2)-1}*'N'(m'iii')*'M'(
#do iiii = 1,'ii'
#if 'iiii' != 'iii'
    m'iiii'
#endif
#enddo
    )
#enddo
    ;
id  'N'(1) = 'n1';
al  'N'(2) = 'n2';
al  'N'(3) = 'n3';
al  'N'(4) = 'n4';
```

```
al  'N'(5) = 'n5';
al  'N'(6) = 'n6';
al  'N'(7) = 'n7';
al  'N'(8) = 'n8';
.sort
#endprocedure
```

The procedure to peel off a determinant has as first argument the size of the minors we are doing (so it starts at 8 in this example), then the name of the function that we use to represent the minors and the determinant. The third argument is a dummy function that we use to determine the matrix elements in each row. Finally there are the 8 elements in the row that we are dealing with. The number 'ii' will be variable so also the number of arguments in M will vary. Therefore we need the preprocessor to generate the argument field of M. We use here the property that blanks are relevant as separators of the arguments. In the right hand side we have to generate an alternating sign for each of the terms, and then the minors which have the same arguments as the original matrix except for one (this is done with a loop again and an if inside the loop to skip one element) and a matrix element in the row we are treating. Finally we substitute these matrix elements. How does this look in practice:

```
S   a0,a1,a2,a3,a4,b0,b1,b2,b3,b4;
S   m1,m2,m3,m4,m5,m6,m7,m8;
CF  M,N;
*
* Matrix is:
*
*     | a0  0  0  0 b0  0  0  0 |
*     | a1 a0  0  0 b1 b0  0  0 |
*     | a2 a1 a0  0 b2 b1 b0  0 |
*     | a3 a2 a1 a0 b3 b2 b1 b0 |
```

```
*     | a4 a3 a2 a1 b4 b3 b2 b1 |
*     |  0 a4 a3 a2  0 b4 b3 b2 |
*     |  0  0 a4 a3  0  0 b4 b3 |
*     |  0  0  0 a4  0  0  0 b4 |
*
L   F = M(1,2,3,4,5,6,7,8);
#call peeldet{8|M|N|a0| 0| 0| 0|b0| 0| 0| 0}
#call peeldet{7|M|N| 0| 0| 0|a4| 0| 0| 0|b4}
#call peeldet{6|M|N|a1|a0| 0| 0|b1|b0| 0| 0}
#call peeldet{5|M|N| 0| 0|a4|a3| 0| 0|b4|b3}
#call peeldet{4|M|N|a2|a1|a0| 0|b2|b1|b0| 0}
#call peeldet{3|M|N| 0|a4|a3|a2| 0|b4|b3|b2}
#call peeldet{2|M|N|a3|a2|a1|a0|b3|b2|b1|b0}
#call peeldet{1|M|N|a4|a3|a2|a1|b4|b3|b2|b1}
id  M() = 1;
print;
.end
```

```
Time =      38.33 sec    Generated terms =        219
                F        Terms left      =        219
                         Bytes used      =       7430
```

Above is the program and the final statistics of the actual run. The order of the rows has be rearranged in such a way that the sign of the determinant didn't change. The variables m1 to m8 were declared because peeldet uses them as dummy variables.

It is actually possible to improve a little bit on this method. On of the drawbacks of the above methods is that the argument field has a length that depends on the size of the matrix. Using sets we can improve on this (as long as we use elements in the sets that are symbols or integers that can be used as powers):

```
#procedure peeldet(ii,nn,M,row)
id 'M'(
```

```
#do iii = 1,'ii'
 m'iii'?
#enddo
 ) =
#do iii = 1,'ii'
 + {2*('iii'%2)-1}*m'iii'*'M'(
#do iiii = 1,'ii'
#if 'iiii' != 'iii'
 m'iiii'
#endif
#enddo
 )
#enddo
 ;
#do iii = 1,'nn'
 id m'iii' = 'row'['iii'];
#enddo
.sort
#endprocedure
S a0,a1,a2,a3,a4,b0,b1,b2,b3,b4;
S m1,m2,m3,m4,m5,m6,m7,m8;
Set row1:a0, 0, 0, 0,b0, 0, 0, 0;
Set row2:a1,a0, 0, 0,b1,b0, 0, 0;
Set row3:a2,a1,a0, 0,b2,b1,b0, 0;
Set row4:a3,a2,a1,a0,b3,b2,b1,b0;
Set row5:a4,a3,a2,a1,b4,b3,b2,b1;
Set row6: 0,a4,a3,a2, 0,b4,b3,b2;
Set row7: 0, 0,a4,a3, 0, 0,b4,b3;
Set row8: 0, 0, 0,a4, 0, 0, 0,b4;
CF M;
L F = M(m1,m2,m3,m4,m5,m6,m7,m8);
#call peeldet{8|8|M|row1}
#call peeldet{7|8|M|row8}
```

```
    #call peeldet{6|8|M|row2}
    #call peeldet{5|8|M|row7}
    #call peeldet{4|8|M|row3}
    #call peeldet{3|8|M|row6}
    #call peeldet{2|8|M|row4}
    #call peeldet{1|8|M|row5}
    id M() = 1;
    .end
```

```
Time =      28.39 sec    Generated terms =        219
                F        Terms left      =        219
                         Bytes used      =       7430
```

We have changed one more thing: the elements of M are now symbols. This makes that we don't need the intermediate 'array' N any more. This is mainly responsible for the speed difference. It is also possible now to write down the matrix in a more readable form.

The results are even more impressive with the 9 by 9 Sylvester determinant that results when a quartic and a fifth order equation are combined. The determinant is then:

```
    S a0,a1,a2,a3,a4,a5,b0,b1,b2,b3,b4;
    S m1,m2,m3,m4,m5,m6,m7,m8,m9;
    Set row1:a0, 0, 0, 0,b0, 0, 0, 0, 0;
    Set row2:a1,a0, 0, 0,b1,b0, 0, 0, 0;
    Set row3:a2,a1,a0, 0,b2,b1,b0, 0, 0;
    Set row4:a3,a2,a1,a0,b3,b2,b1,b0, 0;
    Set row5:a4,a3,a2,a1,b4,b3,b2,b1,b0;
    Set row6:a5,a4,a3,a2, 0,b4,b3,b2,b1;
    Set row7: 0,a5,a4,a3, 0, 0,b4,b3,b2;
    Set row8: 0, 0,a5,a4, 0, 0, 0,b4,b3;
    Set row9: 0, 0, 0,a5, 0, 0, 0, 0,b4;
    CF M;
```

```
    L F = M(m1,m2,m3,m4,m5,m6,m7,m8,m9);
    #call peeldet{9|9|M|row9}
    #call peeldet{8|9|M|row1}
    #call peeldet{7|9|M|row8}
    #call peeldet{6|9|M|row2}
    #call peeldet{5|9|M|row7}
    #call peeldet{4|9|M|row3}
    #call peeldet{3|9|M|row6}
    #call peeldet{2|9|M|row4}
    #call peeldet{1|9|M|row5}
    id M() = 1;
    .end

Time =      86.77 sec    Generated terms =        549
                F        Terms left      =        549
                         Bytes used      =      19690
```

To compute this determinant with the method of the contraction of the Levi-Civita tensors would take of the order of an hour on the same computer, so the savings are rather great. It should be clear by now that with this method even larger determinants can be taken in 'reasonable' amounts of time. The actual timings for two fifth degree equations are 343 sec for this method and 16400 sec for the general method (with some economization so that the order in which the zeroes are substituted is optimized).

27.4 Brute force isn't always good

The following is an example from high energy physics in which the brute force method does work, but with being a little smart we can obtain results that are far superior. We will study a rather common reaction: $e^+e^- \to \tau^+\tau^- \to u\overline{d}\nu_\tau\overline{u}d\overline{\nu}_\tau$. This is a 2 to 6 reaction, but it has some features that make it easier than one might expect. Let us first do it by brute force:

```
V   p1,p2,Q,q1,q2,p3,p4,p5,p6,p7,p8;
I   m1,m2,m3;
I   n1,n2,n3;
S   emass,tmass,mass4,mass5,mass7,mass8;

L   F =
*
*   The incoming e+ e- pair. momenta p2 and p1
*
     (g_(1,p2)-emass)*g_(1,m1)
    *(g_(1,p1)+emass)*g_(1,n1)
*
*   The tau line. tau- is q1, tau+ is q2.
*
    *g_(2,p3)*g_(2,m2)*g7_(2)
    *(g_(2,q1)+tmass)*g_(2,m1)
    *(-g_(2,q2)+tmass)*g_(2,m3)*g7_(2)*g_(2,p6)
    *g_(2,n3)*g7_(2)*(-g_(2,q2)+tmass)*g_(2,n1)
    *(g_(2,q1)+tmass)*g_(2,n2)*g7_(2)
*
*   The u d-bar pair. p4 is u, p5 is d-bar.
*
    *(g_(3,p4)+mass4)*g_(3,m2)*g7_(3)
    *(g_(3,p5)-mass5)*g_(3,n2)*g7_(3)
*
```

```
    *   The d u-bar pair. p7 is d, p8 is u-bar.
    *
        *(g_(4,p7)+mass7)*g_(4,m3)*g7_(4)
        *(g_(4,p8)-mass8)*g_(4,n3)*g7_(4)
        ;
    .sort

Time =       9.41 sec    Generated terms =         64
                F        Terms left      =         64
                         Bytes used      =       5886
    trace4,4;
    trace4,3;
    trace4,1;
    .sort

Time =      12.90 sec    Generated terms =        256
                F        Terms left      =        256
                         Bytes used      =      14354
    trace4,2;
    contract 0;
    *
    *   We kill the leftover Levi-Civita tensors. We
    *   know that there must be a relation that makes
    *   them cancel each other because the reaction
    *   is time reversal invariant:
    *
    if ( count(e_,1) > 0 );
        discard;
    endif;
    .end

Time =    1257.49 sec    Generated terms =     226280
                F        Terms in output =       2848
```

```
                    Bytes used      =     147316
```

We have an output here that cannot be considered small and continuing numerical work with such a formula will be time consuming. So let us have a look at some of the individual parts. First the electron trace:

```
V   p1,p2,Q;
I   m1,m2,m3;
I   n1,n2,n3;
S   emass,tmass,mass4,mass5,mass7,mass8,s;

L   F =
         (g_(1,p2)-emass)*g_(1,m1)
        *(g_(1,p1)+emass)*g_(1,n1)
    ;
trace4,1;
print;
.sort

F =
   4*p1(m1)*p2(n1) + 4*p1(n1)*p2(m1) - 4*d_(m1,n1)
   *emass^2 - 4*d_(m1,n1)*p1.p2;
*
*   Use momentum conservation. Q = p1 + p2
*
id  p2 = Q - p1;
id  p1.p1 = emass^2;
id  Q(m1) = 0;
al  Q(n1) = 0;
id  p1.Q = Q.Q/2;
print;
.end
```

```
F =
    - 8*p1(m1)*p1(n1) - 2*d_(m1,n1)*Q.Q;
```

The first step gives the brute force result with 4 terms. Some massaging and the observation that current conservation will give that `Q(m1)=Q(n1)=0` reduces things to two terms. This would make the computation faster and the result shorter.

The next step is more interesting. We observe that the decay vertices both have a similar structure. We take here the $u\overline{d}$ part with a little part of the long tau line. Of course we may not take the trace over spin line 2, but we can 'Fierz' it. This is done with the multiply command and the subsequent trace.

```
V   p1,p2,Q,q1,q2,p3,p4,p5,p6,p7,p8;
I   m1,m2,m3;
I   n1,n2,n3;
S   emass,tmass,mass4,mass5,mass7,mass8;

L   F =
    g_(2,n2)*g7_(2)*g_(2,p3)*g_(2,m2)*g7_(2)
    *(g_(3,p4)+mass4)*g_(3,m2)*g7_(3)*(g_(3,p5)
    -mass5)*g_(3,n2)*g7_(3)
    ;
trace4,3;
print;
.sort

F =
   16*g_(2,6_,p4,p3,p5) + 16*g_(2,6_,p5,p3,p4) +
   16*g_(2,6_,n2,p3,m2)*e_(p4,p5,m2,n2) - 16*g_(2,
   6_,n2,p3,n2)*p4.p5;

Symbols as,ap;
Vectors av,aa;
```

```
CFunction at;
*
*   Fierz transformation:
*
multiply,
     as*gi_(2)/4
    +ap*g5_(2)/4
    +g_(2,av)/4
    -g_(2,5_,aa)/4
    -at(m1,n1)*(g_(2,m1,n1)-g_(2,n1,m1))/8;
trace4,2;
contract;
id  as = gi_(2);
al  ap = g5_(2);
al  av = g_(2,?);
al  aa = g_(2,5_,?);
al  at(m1?,n1?) = g_(2,m1,n1)/2-g_(2,n1,m1)/2;
print;
.end

F =
   64*g_(2,p5)*p3.p4 + 64*g_(2,5_,p5)*p3.p4;
```

Note the great economization. First we had 4 terms with three gamma matrices and one had a Levi-Civita tensor. The Fierz transformation simplified the expression considerably! The only drawback that occurs is that the $1 - \gamma_5$ is not a single object any more. This can be mended with a single id-statement of course. Now the final program:

```
V   p1,p2,Q,q1,q2,p3,p4,p5,p6,p7,p8;
I   m1,m2,m3;
I   n1,n2,n3;
S   emass,tmass,mass4,mass5,mass7,mass8;
```

```
Symbols as,ap;
Vectors av,aa;
CFunction at;
.global
G   F1 =
    (g_(1,p2)-emass)*g_(1,m1)
    *(g_(1,p1)+emass)*g_(1,n1);
trace4,1;
id  p2 = Q - p1;
id  p1.p1 = emass^2;
id  Q(m1) = 0;
al  Q(n1) = 0;
id  p1.Q = Q.Q/2;
.store
G   F3 =
    g_(2,n2)*g7_(2)*g_(2,p3)*g_(2,m2)*g7_(2)
    *(g_(3,p4)+mass4)*g_(3,m2)*g7_(3)
    *(g_(3,p5)-mass5)*g_(3,n2)*g7_(3);
G   F4 =
    g_(2,m3)*g7_(2)*g_(2,p6)*g_(2,n3)*g7_(2)
    *(g_(4,p7)+mass7)*g_(4,m3)*g7_(4)
    *(g_(4,p8)-mass8)*g_(4,n3)*g7_(4);
trace4,3;
trace4,4;
multiply,
     as*gi_(2)/4
    +ap*g5_(2)/4
    +g_(2,av)/4
    -g_(2,5_,aa)/4
    -at(m1,n1)*(g_(2,m1,n1)-g_(2,n1,m1))/8;
trace4,2;
contract;
id  as = gi_(2);
```

```
    al  ap = g5_(2);
    al  av = g_(2,?);
    al  aa = g_(2,5_,?);
    al  at(m1?,n1?) = g_(2,m1,n1)/2-g_(2,n1,m1)/2;
    id  g_(2,5_,p5?) = g_(2,p5)*g7_(2) - g_(2,p5);
    print;
    .store

   F3 =
      64*g_(2,6_,p5)*p3.p4;

   F4 =
      64*g_(2,6_,p7)*p6.p8;

    write statistics;
    L   F = F1 *
         F3*(g_(2,q1)+tmass)
        *g_(2,m1)*(-g_(2,q2)+tmass)
        *F4*(-g_(2,q2)+tmass)
        *g_(2,n1)*(g_(2,q1)+tmass)
        ;
    trace4,2;
    print;
    .end

Time =       1.77 sec    Generated terms =         41
                F        Terms left      =         23
                         Bytes used      =       1202

   F =
      262144*p1.p1*q1.q1*q2.q2*p3.p4*p5.p7*p6.p8 -
      524288*p1.p1*q1.q1*q2.p5*q2.p7*p3.p4*p6.p8 +
      1048576*p1.p1*q1.q2*q1.p5*q2.p7*p3.p4*p6.p8 +
```

```
524288*p1.p1*q1.q2*p3.p4*p5.p7*p6.p8*tmass^2 -
524288*p1.p1*q1.p5*q1.p7*q2.q2*p3.p4*p6.p8 +
524288*p1.p1*q1.p5*q2.p7*p3.p4*p6.p8*tmass^2 -
524288*p1.p1*q1.p7*q2.p5*p3.p4*p6.p8*tmass^2 +
262144*p1.p1*p3.p4*p5.p7*p6.p8*tmass^4 -
2097152*p1.q1*p1.q2*q1.p5*q2.p7*p3.p4*p6.p8 -
1048576*p1.q1*p1.q2*p3.p4*p5.p7*p6.p8*tmass^2
 + 1048576*p1.q1*p1.p7*q1.p5*q2.q2*p3.p4*p6.p8
 + 1048576*p1.q1*p1.p7*q2.p5*p3.p4*p6.p8*
tmass^2 + 1048576*p1.q2*p1.p5*q1.q1*q2.p7*p3.p4
*p6.p8 + 1048576*p1.q2*p1.p5*q1.p7*p3.p4*p6.p8*
tmass^2 - 524288*p1.p5*p1.p7*q1.q1*q2.q2*p3.p4*
p6.p8 - 1048576*p1.p5*p1.p7*q1.q2*p3.p4*p6.p8*
tmass^2 - 524288*p1.p5*p1.p7*p3.p4*p6.p8*
tmass^4 + 131072*Q.Q*q1.q1*q2.q2*p3.p4*p5.p7*
p6.p8 - 262144*Q.Q*q1.q1*q2.p5*q2.p7*p3.p4*
p6.p8 + 524288*Q.Q*q1.q2*q1.p5*q2.p7*p3.p4*
p6.p8 - 262144*Q.Q*q1.p5*q1.p7*q2.q2*p3.p4*
p6.p8 + 524288*Q.Q*q1.p5*q2.p7*p3.p4*p6.p8*
tmass^2 + 131072*Q.Q*p3.p4*p5.p7*p6.p8*tmass^4;
```

Note the enormous economization. The tricks we employed with F3 and F4 removed all Levi-Civita tensors and eliminated all ambiguities in the final form of the expression. This final form hardly took any computer time and is only 23 terms long! It is actually possible to reduce this output a little further but that isn't very important any more. The above formula is quite useful as it contains not a single approximation. Neither the electron mass nor the quark masses have been ignored. Actually even a mass for the neutrino would not have upset this result, as this neutrino mass would have disappeared in the Fierz operation. We have not even assumed that the τ leptons are on shell!

The above computation shows several things. First of all it pays to think when brute force doesn't work. Second of all there is much room for different answers when there are Levi-Civita tensors involved and there are more vectors than the number of dimensions. This is illustrated in the following program that operates in two dimensions:

```
Dimension 2;
Vectors p1,p2,p3,p4,p5,p6;
Indices m1;
Symbols a;
L   F1 = e_(p1,p2)*e_(p3,m1)*e_(p4,m1)*e_(p5,p6);
L   F2 = e_(p1,p2)*e_(p3,m1)*a;
contract;
id  a = e_(p4,m1)*e_(p5,p6);
contract;
print;
.end

F1 =
   p1.p5*p2.p6*p3.p4 - p1.p6*p2.p5*p3.p4;

F2 =
    - p1.p3*p2.p5*p4.p6 + p1.p3*p2.p6*p4.p5 +
   p1.p5*p2.p3*p4.p6 - p1.p6*p2.p3*p4.p5;
```

In the first expression FORM selected the ideal contraction scheme by taking the tensor with a common index first. In the second expression this wasn't possible. Nevertheless both expressions are equal, even though this may not be very clear at first sight. This is what happened in a much more complicated fashion when we ran the brute force method. The Levi-Civita tensors couldn't be combined in the 'natural' way. In the second method

the Fierz operation took care that they did. Actually in the last example the difference F1-F2 can be written as:

```
e_(p1,p2,p4)*e_(p3,p5,p6)
```

which is of course zero because we work in two dimensions and any totally antisymmetric object with three indices must be zero then. To find such relations is usually nontrivial and no reliable way has been found to construct an automatic implementation in computer algebra.

Index